ENABLING MODULAR APPLICATION DEVELOPMENT FOR MANAGEMENT AND SECURITY IN SOFTWARE-DEFINED NETWORKS

A Dissertation
Presented to
The Academic Faculty

By

Jacob H. Cox Jr.

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2017

ENABLING MODULAR APPLICATION DEVELOPMENT FOR MANAGEMENT AND SECURITY IN SOFTWARE-DEFINED NETWORKS

Approved by:

Dr. Owen, Henry
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Clark, Russell
College of Computing
Georgia Institute of Technology

Dr. Beyah, Raheem
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Copeland, John
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ahamad, Mustaque
College of Computing
Georgia Institute of Technology

Date Approved: March 1, 2017

Whatever you do in word or deed, do all in the name of the Lord Jesus, giving thanks through Him to God the Father.

Colossians 3:17

To Christ with whom my salvation lies,

To my wife, Laura, for her love and support and whom I adore, and

To my parents who loved me without condition and provided all I have ever needed

ACKNOWLEDGEMENTS

This dissertation would have not been possible without all the support, guidance, and encouragement I have received along the way. Specifically, I owe my advisors, Professor Henry Owen and Dr. Russell Clark, the United States Army, my parents, and my wife a huge debt of gratitude for making this opportunity possible.

Much thanks is owed to my advisor, Professor Henry Owen, for offering his guidance and support even before I arrived at Georgia Tech to begin my studies. He fully understood my needs as a military officer, and he provided the essential guidance I needed to successfully navigate the requirements leading to a Ph.D. Furthermore, his advice to immediately attempt the preliminary exam and encouragement throughout my preparations pushed me to quickly overcome the exam and transition to my research.

I am likewise grateful to Dr. Russell Clark, who served as my co-advisor, offering both his support and encouragement to complete my academic research. Beyond frequently reviewing my work, his support provided me with much appreciated opportunities to gain greater familiarity with cyber infrastructure, cyber security, and cloud architectures.

Much thanks is also due the faculty at the Department of EECS, United States Military Academy, West Point. I am deeply appreciative of COL Lisa Shay, COL Gregory Conti, COL(R) Scott Ransbottom, COL Barry Shoop, and so many other senior faculty who placed their confidence in me to complete this program. Their advice and mentoring set me up for success throughout this endeavor.

I also remain grateful for the pride and love of my parents. Jessie and Thomas Crongeyer, Jacob and Lana Cox, and Michael and LaVernna Kapica remain blessings in my life.

Finally, I will always be gratefully indebted to my wife, Laura, whose unwavering support and love throughout my life and career has encouraged me to grow and excel. Whether together or apart, she always strives to make me feel special and appreciated. After 20 years of marriage, I love her all the more.

TABLE OF CONTENTS

Acknowledgments v

List of Tables xi

List of Figures xii

Chapter 1: Introduction 1

1.1 Towards Simplifying SDN Management and Security 4

1.2 Contribution 4

1.3 Background 6

1.3.1 Software Defined Networking 7

1.3.2 Network Functions Virtualization 8

1.3.3 Network Security 9

1.3.4 Challenges To Network Administration 11

1.3.5 SDN Application Development for Management and Security . . . 12

1.4 Technical Approach 13

1.4.1 Outline 15

1.5 Bibliographic notes 19

Chapter 2: Leveraging SDN to Improve the Security of DHCP 20

2.1 Introduction 20
2.2 State of the Art 21
2.3 Related Work 23
2.4 Design and Implementation 24
2.4.1 NFG Design 24
2.5 Test Environment and Results 27
2.6 Discussion 29
2.7 Future Work 30
2.8 Conclusion 31

Chapter 3: Leveraging SDN for ARP Security 32

3.1 Introduction 32
3.2 Background 33
3.3 State Of The Art 36
3.4 Design and Implementation 39
3.4.1 NFG Design 39
3.4.2 Implementation 43
3.5 Test Environment and Results 45
3.6 Related Work 47
3.7 Discussion 49
3.8 Future Work 50
3.9 Conclusion 51
3.10 Segue 52

Chapter 4: Ryuretic: A Modular Programming Framework for Ryu 53

4.1 Introduction 53

4.2 The Ryuretic Programming Framework 55

4.3 Ryuretic Components 58

4.3.1 Coupler 59

4.3.2 Packet Parser 62

4.3.3 Switch Module 63

4.4 Ryuretic Programming Examples 64

4.4.1 Simple Stateful Firewall Application 64

4.4.2 Unauthorized NAT device 66

4.5 Discussion 68

4.6 Future Work 69

4.7 Conclusion 70

4.8 Segue 70

Chapter 5: Security Policy Transition Framework for Software-defined Networks 71

5.1 Introduction 71

5.2 Framework Motivation 72

5.3 Related Work 74

5.4 The Framework 76

5.4.1 Controller 77

5.4.2 Trusted Agent 79

5.4.3 Communication Channel 81

5.5 Test Environment 83
5.6 Example Use Case 83
5.6.1 Spoofed ARP Packets 83
5.7 Discussion and Future Work 85
5.8 Conclusion 86
5.9 Segue 87

Chapter 6: Leveraging SDN and WebRTC for Rogue Access Point Security . . . 88
6.1 Introduction 88
6.2 Background 90
6.2.1 Network Address Translation (NAT) 91
6.2.2 Rogue Access Points (RAP) 91
6.3 Rogue Device Detection Methods 92
6.3.1 Client-side approach 92
6.3.2 Server-side approach 94
6.4 Network Flow Guard Security 98
6.4.1 Requirements and Assumptions 98
6.4.2 Security Features 99
6.5 Network Flow Guard Architecture 106
6.5.1 SDN Controller 108
6.5.2 Trusted Agent 109
6.5.3 Communication Channel 111
6.6 Test Environment and Results 112

6.7 Discussion 120
6.8 Future Work 122
6.9 Conclusion 123

Chapter 7: Conclusion 124
7.1 Other Results 124
7.2 Discussion 126
7.3 Contributions 130
7.4 Future Work 131
7.5 Conclusion 133

Appendix A: Experimental Equipment 135
A.0.1 Hardware 135
A.0.2 Virtual Environment 135
A.0.3 Tools 136

References 149

Vita 150

LIST OF TABLES

3.1 ARP Detection and Protection Tools [55] 37

4.1 Pyretic Fields 53

4.2 Ryu Fields 54

4.3 Ryuretic Fields: fields[‘*’]. 57

4.4 Ryuretic Operations: ops[‘*’]. 57

4.5 Ryuretic Pkt Object: pkt[‘*’] 63

5.1 Abbreviations Used for Controller Communication [30] 82

6.1 Server Side NAT Detection Techniques 93

6.2 RAP Characteristics by Type 100

6.3 Controller Communication Abbreviations 111

LIST OF FIGURES

1.1 Evolution of dissertation concepts 14

2.1 NFG Implementation. 25

2.2 NFG Flowchart for DHCP Traffic. 27

3.1 ARP Request/Reply Packet Sequence 34

3.2 Normal network flow. 35

3.3 Man-in-the-middle. 35

3.4 Denial of Service (DOS) or Server Redirect 35

3.5 High level view of NFG Operation 40

3.6 NFG ARP Validation Flow Graph 41

3.7 Dynamic Table Entry Steps 41

3.8 DHCP Packet Flow Graph for Dynamic Table 42

3.9 NFG Implementation 43

3.10 Pyretic Query used by NFG 44

3.11 Dynamic Table 46

4.1 Modules Joined Via the Ryuretic Coupler 55

4.2 Ryuretic Framework 56

4.3 Ryuretic Packet Builder 56

4.4 Ryuretic Controller 58

4.5 Stateful Firewall Event Sequence 64

4.6 Ryuretic Labs 69

5.1 Security Policy Transition Framework 73

5.2 Security Policy Transition Framework Components [30] 77

5.3 Ryuretic Controller 78

5.4 ICMP Packet Header 81

5.5 Controller - Trusted Agent Communication 82

6.1 Network Flow Guard's RAP Detection Flowchart 99

6.2 State Diagram for Capture of Port Flow TTA 102

6.3 Port Interception on SDN Switch 104

6.4 NFG RAP Detection Framework 106

6.5 Controller - *Trusted Agent* State Information 107

6.6 Mininet-WiFi Implementation and Testbed 112

6.7 RTT from clients to gateway router 113

6.8 Rolling port averages at varying weights and seed of 5 msec 114

6.9 Rolling port average with weight of 9 and seed of 5 msec 114

6.10 TTA analysis of iperf results per port 116

6.11 TTA analysis of curl script results per port 116

6.12 Rolling average analysis of curl script results per port 117

6.13 WebRTC Results (Firefox Web Browser) 118

6.14 Server BW comparison of Ryu vs Ryuretic 119

6.15 Client BW comparison of Ryu vs Ryuretic 119

7.1 Code comparison of Ryuretic and Ryu based Stateful Firewalls 124

7.2 Code comparison of Ryuretic and POX and based Stateful Firewalls 125

SUMMARY

This dissertation leverages the capabilities of software-defined networking (SDN) and network functions virtualization (NFV) to enhance network security and management. By first exploring SDN-based security solutions and then systematically building an SDN-based programming framework and a security policy transition framework, this research makes possible a security/management system for SDNs that is also capable of reducing network operator workloads. With this work's programming framework, Ryuretic, network operators are offered more intuitive abstractions for creating their own network applications using fewer lines of code. Additionally, network operator configuration requirements are reduced by the incorporation of an automated security policy transition framework, enabled through NFV, which automatically updates or revokes policy enforcements–subsequently helping to reduce human errors on the network. Together, these features allow network operators to create complete security/management solutions that incorporate both passive and active network testing methods into an automated system for managing the state transitions of policy enforcements on software-defined networks.

CHAPTER 1
INTRODUCTION

While network functions virtualization (NFV) is an established technology for many organizations, software-defined networking (SDN) is an emerging paradigm that is steadily seeping into campus, government, and industry networks. As a result, SDN controllers are quickly maturing to offer greater abstractions and more intuitive programming frameworks for network operators seeking to develop their own network applications. Additionally, with new network vulnerabilities emerging and exploitations continuing to occur, even as the world grows ever more connected, these organizations are placing greater priority on network security in order to protect their clients, data, and resources. Hence, SDN-based and NFV-based security solutions are an ever growing commodity.

Because SDN separates the network's control logic from its underlying switches and routers (i.e., the data plane), many of SDN's proponents argue that it provides network operators greater flexibility, higher levels of abstraction, and rapid development capabilities for new network applications. As a result, network management and network evolution are greatly simplified under a logically centralized and programmable network. Accordingly, it is the desire for network programmability and abstractions for network management and security that forms the underlying foundation for this work–particularly along network edge-devices.

In contrast, the tight coupling of the control and data plane within traditional network devices makes traditional network application development more tedious, error-prone, and expensive to manage. For instance, within traditional environments, network operators enter combinations of low-level commands to update configuration files that reside on individual network devices. As we will highlight in Chapter 5, such configurations can occur as many as 1,000 to 18,000 times per month [1, 2]. Of course, with such repeated and manual

configuration requirements, network operators can frequently introduce new problems to their network via human error. Accordingly, Benson et al. [3] observe that a large percentage of network outages are the result of configuration errors, which increase with growing network complexity. Moreover, such configurations absorb additional cost (e.g., money, time, availability, etc.) as network operators must first implement specified requirements via command line interface (CLI) and then commit additional time to troubleshooting errors that may not be immediately visible after the configuration is complete.

Additionally, due to the specialized and proprietary nature of the vendor software that resides on traditional network devices, significant cost is associated with purchasing, maintaining, and evolving such devices. Furthermore, waiting on vendors to modify their proprietary boxes to accommodate new needs can often take months to years, and that is if the vendor agrees to add the desired features at all [4]. In this regard, SDN and NFV offer significant advantages to network operators and researchers in terms of rapid network prototyping, equipment reuse, and the incorporation of new network protocols and security features.

Accordingly, the motivations for turning to SDN and NFV for better network management and security are many. Their benefits include reduced costs, ease of deployment and management, better scalability, availability, flexibility, and fine-grained control of traffic and security. Thus, we explore various security challenges and benefits of SDN, while also providing novel methods and approaches that also incorporate NFV to provide better security and policy management in SDN-based networks. Hence, we argue that SDN with NFV can better empower network operators in the defense and management of their networks with better tools and abstractions that allows them to rapidly address an ever evolving network attack landscape.

Still, this work does not attempt to justify SDN as a better alternative to traditional networks. In truth, we simply observe that SDN and NFV are gaining acceptance by a growing number or organizations. Many of which, have already incorporated SDN and

NFV into their network infrastructure. We also highlight that many of the network solutions discussed in this dissertation have already been addressed, or could be addressed, using traditional and proprietary solutions. In such cases, we highlight the state-of-the-art for what has been done within traditional networks (and within SDNs when applicable) before discussing our own solutions. In doing so, we evaluate several security challenges that continue to plague network operators on traditional networks.

Yet, while these solutions often detect and block clients who violate network policies, they frequently fail to consider how policy enforcements will be revoked or updated. Such instances occur when a flagged client addresses a violation for which they are flagged or when a client is flagged for testing and must next transition to a 'deny' or 'allow' state. As it happens, no clear path exists for a client's re-instantiation to the network beyond having the network operator manually remove the policy enforcement or reset the SDN controller. Likewise, the network operator is often involved in the process of transitioning a flagged client from a state of 'test' to a state of 'allow' or 'deny', or some other combination (e.g., transitioning from a state of 'deny' back to 'allow'). As previously stated, these requirements are tedious and error-prone. Additionally, these efforts cost valuable time that could be better utilized for more complex network tasks. Hence, this dissertation offers a security policy transition framework for reducing wait times and automating the revocation of policy enforcements in SDN environments for clients who are approved to rejoin the network.

As previously indicated, the applications developed in this work are intended for edge-devices. Hence, they are intended to serve as a first line of defense in a defense-in-depth security strategy for network operators. Likewise, these solutions also seek to limit the network operator's involvement in the setting, updating, and removing of policy enforcements through the incorporation of an automated security policy transition framework. Furthermore, it builds off of this framework to offer active security testing methods that are also automated through this transition framework. As such, the solutions offered in this disser-

tation are not considered for application within the higher tiers of network infrastructure. Instead, the developed applications of this work serve to aid network operators with network management while also providing solutions for an initial barrier to network attackers. We also contend that end devices (i.e., hosts) are not trustworthy resources for the applications developed in this work. Likewise, we do not believe host-based security solutions to be scalable or maintainable by network operators. As a result, this work, with the exception of its Trusted Agent, does not require host cooperation as do other security systems. Nor does our work require changes to network architecture since security is implemented at the switch via its controller. Moreover, our work is fully adaptable to cloud environments and all components and functions of our work can easily be replicated via network functions virtualization.

1.1 Towards Simplifying SDN Management and Security

Thesis Statement: Given that many organizations are embracing the software-defined networking (SDN) paradigm in conjunction with network functions virtualization (NFV), this work posits that frameworks for application development and policy management are needed and explores the capabilities of SDN and NFV to better assist network operators with creating network applications for network policy management and security. Accordingly, we use SDN, aided by NFV, to offer a modular, programming framework and a security policy transition framework to empower network operators with better tools and abstractions to reduce manual network tasks and address evolving network attack vectors. Within this system, we also introduce the concept of a Trusted Agent for assisting the SDN controller with setting, updating, and revoking policy enforcements and with active testing.

1.2 Contribution

This dissertation makes the following contributions towards the defense of its thesis statement. They include the development of a modular programming language and a security

policy transition framework, the introduction of a Trusted Agent, the development of an in-band communication technique for the SDN controller, and a novel method for detecting rogue access points on a network using the webRTC architecture. These contributions are now discussed.

Develops a modular, programming framework for the RYU controller. This dissertation introduces Ryuretic [5], which is a domain specific language (DSL) for creating modular applications atop the RYU controller. It offers high-level abstractions to simplify network application development. Using Ryuretic's abstractions, network operators need only understand basic programming concepts to create robust network applications. Accordingly, this dissertation offers several modular network security and policy applications that promote code reuse. Likewise, there is a simple and intuitive process for adding network and network security modules to the Ryuretic interface that better enables network operators to share their network enhancements with others. Moreover, Ryuretic offers access to over 40 packet field headers, while rivals such as Pyretic and Kinetic offer access to only 12.

Explores the use of a Trusted Agent to enhance SDN capabilities. Using NFV, we introduce a Trusted Agent to facilitate both policy transitions and active testing within an SDN. While most SDN security applications focus on blocking ports of offending clients, little thought is given to reinstating the client in an automated manner once the client gains approval to rejoin the network. In this dissertation, we utilize a Trusted Agent as part of a network policy system that not only detects/blocks/redirects unwanted user traffic, but also leads the user through a work flow to regain normal access. Additionally, the Trusted Agent works hand-in-hand with the SDN controller to perform active detection on clients who have been flagged by SDN applications for possible policy violations.

Offers the first SDN application to detect and deny rogue access points. While this dissertation develops several applications that exist on traditional networks for detecting rogue devices, including a couple for detecting rogue access points, it also offers a novel use

of the webRTC architecture to detect subnets. Using developing APIs for web architectures, the capabilities of SDN to intercept ports as they open (port-intercepting), and a Trusted Agent to render webRTC scripts, this work demonstrates a new way to detect subnets and reveal network address translation devices and rogue access points (i.e. wireless routers).

Develops an SDN to Trusted Agent communication protocol. One limitation of SDN controllers is they only receive a packet's header information. Likewise, unless the controller has an east-west bound interface, communicating with the controller is not possible. In this work, we develop a limited communication protocol allowing for the controller to communicate with its Trusted Agent for policy adjustments and testing requirements.

1.3 Background

The tight coupling of the control and data plane within traditional network devices makes security solutions tedious, time consuming, and prone to error [6]. On such networks, operators are often required to interact with each individual device via a command line interface (CLI) to emplace even the simplest network modifications [7]. Moreover, when implementing security features, such efforts fail to capitalize on the header information and state already available to edge-devices, and instead force network operators to rely on proprietary solutions such as specialized middleboxes or a vendor specific CLI to address issues on a per port basis. SDN holds tremendous promise to address many network challenges. For instance, SDN is expected to provide lower cost hardware, faster upgrade cycles, lower OPEX, and lower energy costs [8]. Additionally, SDN is already being leveraged for new products and features, including security. Moreover, major industry leaders in telecommunications, digital advertising, and data are embracing this technology. Yet, even as these organizations move towards SDN as a means to reduce errors and simplify network management, handling policy enforcements and mitigating network operator configurations remains a problem.

In some cases, the configuration burden is just moved from one format to another as

organizations move from traditional network infrastructure to SDN. The result being that the network operator is still responsible for addressing policy transitions within an SDN environment. This can create significant and tedious administration challenges for network operators. To highlight these issues and motivate the need for SDN-based management and security solutions, the following sections highlight administration challenges, SDN capabilities with NFV as an enabler, application development, and security challenges.

1.3.1 Software Defined Networking

In contrast to traditional networks, a Software-Defined Network (SDN) separates the control and data planes, so network intelligence and state are abstracted away and maintained by a logically centralized controller [9, 6]. Simply said, SDN decouples control logic from vendor-specific hardware [10]. Likewise, the controller is abstracted away from network applications [9]. As a result, network operators are able to create applications atop the core services of a network operating system (NOS) instead of its underlying infrastructure [11]. This process is much akin to how programmers create programs for computer operating systems. The SDN paradigm also makes it possible for researchers to leverage OpenFlow [12] switches to handle various security threats.

Already, SDN solutions have been implemented to detect and mitigate various attack vectors like port scans [13], denial of service (DoS) [14] attacks, and anomalies [15, 13]. Unfortunately, none of these solutions offer a modular approach to implementing multiple security solutions. Nor are they designed to work with current network protocols. As such, they allow little room for augmentation, or they are not immediately adaptable by network operators. However, if approached properly, the above security solutions could also be included in an SDN-based security framework (e.g., Network Flow Guard) as modular, add-on components that allow network operators to choose the features best suited to their networks without being relegated to a CLI for its implementation. Accordingly, this work capitalizes on two SDN frameworks. They are Pyretic [16], a python-based, modular,

programming language for SDN, and Ryu [17], a component-based SDN framework.

SDN has also already been used to successfully solve such important problems as cost, management, multi-tenancy, and high-entry barriers that limit innovation. As of 2015, 12% of industry and 17% of Data Centers are already deploying SDNs [18, 19]. Some of the reasons for SDN's growing popularity include its integration of network fabrics and external cybersecurity platforms to offer greater network visibility along with the enforcement of domain-specific real-time policies.

Since SDN allows network operators to implement network applications at a higher level of abstraction with the SDN controller handling packet decisions [20, 16], SDN solutions can potentially replace existing hardware solutions (e.g., middleboxes). While middleboxes provide security, performance, and policy compliance benefits, they also require considerable operator expertise and manual effort to deploy [21]. Nor is it trivial to determine the best means to direct traffic to appropriate middleboxes [21]. Resultantly, Sherry et al. [22] observe that the number of middleboxes on today's networks are roughly equal to the number of routers and switches found on the same networks, regardless of size. Hence, this work's Network Flow Guard (NFG) framework provides the added benefit of removing some middlebox requirements.

1.3.2 Network Functions Virtualization

While network functions vitalization (NFV) is not a primary focus of this work, it is a key enabler for components, servers, and network devices implemented in this dissertation. NFV serves to separate network functions from the physical devices on which they run and is a key enabler for the components utilized throughout this dissertation. The use of NFV brings with it significant potential to reduce operating expenses (OPEX) and capital expenses (CAPEX) and facilitates the rapid deployment of new and agile services [23]. Likewise, researchers can use NFV to develop new architectures, systems, and applications to further evaluate developing technologies and better understand their trade-offs, capabil-

ities, and alternatives.

Traditionally, network operators have deployed physical proprietary devices and equipment in order to achieve specific network functions (e.g., firewalls, load balancers, NAT, IDS, IPS, etc.). These devices have often come with strict chaining and/or ordering that limits network topologies with localized services that possess little agility and a strong dependence on specialized, vendor-centric hardware solutions [23]. However, with networks growing all the more diverse and requiring faster data rates, network operators often find themselves in a short and repeating cycle of purchasing, storing, and operating new physical equipment while replacing and storing or destroying old equipment. Additionally, even minor changes for a given function can require equipment replacement.

Hence, NFV adds value to this work by offering a cheap and flexible way to design, deploy and manage networking services in a test environment. Using Linux-based containers in a Mininet [24] or Mininet-WiFi [25] environment, this work implements a variety of network functions: Dynamic Host Configuration Protocol (DHCP), Network Address Translation (NAT), Wireless Access Point (WAP), web server, SDN controller, Open vSwitch, rogue Client, Trusted Agent, etc. SDN is then used to provide orchestration to the switch that interconnects these various virtual network functions (VNFs). As a result, this dissertation is applicable to SDN-NFV research communities.

1.3.3 Network Security

Like its predecessor, work in SDN has moved forward with little regard to security. This has led some researchers to claim that the state of network security is abysmal [26]. As a result, this area of research continues to present vast opportunities for researchers. It also leaves much to be desired by network operators. SDN security essentially falls into two categories.

First is the security of the SDN architecture itself. One can imagine a "stuxnet-like" [27] virus designed specifically to hijack, shutdown, or corrupt SDN controllers. The impact that

such a virus could have on an SDN is indeed significant. Some researchers already argue that one weakness of SDN that impacts its commercial adoption is its inability to offer security and dependability [28]. However, SDN is not alone with its security challenges. For instance, traditional networks, have their own issues, and recent reports indicate that the control plane of traditional networks is no longer as safe as once thought [29]. The second category focuses instead on detecting and preventing malicious network activity. Such activity can include any number of attacks capable of affecting or accessing the network's resources, services, and/or clients.

The work presented in this dissertation falls squarely into the latter category. Its solutions work to detect network attacks that have plagued traditional networks (e.g., rogue DHCP servers, ARP poisoning, rogue access points, etc.) and continue to be exploited. Of course, Cyber-attacks are of great concern across all networks and they continue to push the boundaries of both traditional networks and SDNs. The advantage of an SDN however is that new security solutions can be rapidly prototyped and deployed without need for vendor buy-in. So, organizations interested in detecting and denying a number of other attack vectors, including advanced persistent threats (APTs), data exfiltration, malware propagation, denial of service, etc., can capitalize on SDN's open source features to hasten the development of security solutions.

Hence, this work provides a framework capable of supporting the inclusion of such security measures into its framework as modular applications. Additionally, this work leverages the capabilities of SDNs to exploit the state and header information available to edge-devices on local networks in order to mitigate or eliminate existing attack vectors and reduce manual network configuration changes. For instance, Network Flow Guard DHCP (NFGD) introduces an extensible module for detecting and preventing Rogue DHCP servers (see Chap. 2), and Network Flow Guard ARP (NFGA) introduces a similar module for detecting and preventing ARP poisoning (see Chap 3). To handle more complex security solutions Chap. 4 introduces Ryuretic [5] as a new programming framework for

SDNs, and Chap. 5 introduces a security policy transition framework [30]. And, when combined, the previous two chapters enable the creation of a security system for detecting and denying rogue access points (RAPs) using both passive and active measures as we will discuss in Chap. 6.

In spite of the solutions offered in this dissertation, its developed frameworks are only considered as a singular piece of an overall security strategy for networks. Instead, we offer it as first defense in a defense-in-depth strategy. Likewise, it has little applicability towards network security beyond local subnets. In fact, moving this framework to higher network tiers (beyond edge-devices) would thwart some of the security solutions implemented in this work as we will discuss in Chap. 7.

1.3.4 Challenges To Network Administration

Computer networks are large and complex, many consisting of over 1,000 network devices (e.g., routers, switches, firewalls, IDPS, and multiple other middleboxes). In spite of its complexity and size, network configuration still relies largely on manual configurations. Even within SDN environments, network operators may find themselves repeatedly interacting with SDN controllers in order to reset the policy enforcements that are activated by security policies. As a result, they need abstractions to simplify the development of network applications and to enforce policies. Likewise, network operators need frameworks that reduce their involvement with network policy enforcements and network configuration changes. And, where possible, they can benefit from an ability to pick and choose which network modules they wish to include in their SDN.

In this dissertation, we first demonstrate the capabilities of a modular programming framework to couple applications together to enforce multiple network policies and improve network security. Next, due to the limitations of current, modular programming languages, we offer a programmable framework to overcome existing limitations of programming languages (e.g., Pyretic [16] and Kinetic [2]) in order to introduce new options

for network application development to network operators. Within this framework, called Ryuretic [5], network operators can forward, drop, redirect, mirror, modify, and even craft packets in a simple and intuitive manner. Additionally, the Ryuretic framework allows network operators to create fine-grained network applications that target specific layers of the OSI model. Once this programming framework is discussed, we then introduce a security policy transition framework for SDNs that seeks to reduce network operator configuration requirements through automation of policy enforcement revocations by using a Trusted Agent. The Trusted Agent serves as an intermediary between the network operator, the client, and the SDN controller and facilitates the transition of policy enforcements. Finally, this work demonstrates how the Ryuretic programming framework, working in concert with a Trusted Agent, can not only automate the revocation of policy enforcements, but also expand the capabilities of SDN through NFV to provide active detection methods and enhance network security.

1.3.5 SDN Application Development for Management and Security

One of the goals of this work is to minimize the SDN controller's involvement with non-routing/security related tasks. For instance, where some researchers have allowed the SDN controller to actively perform all aspects of a network protocol, such as replying to DHCP and ARP requests, this work seeks to allow current protocols to go unmolested by the controller. Such functions are easily implementable as virtual network functions (VNFs) if needed, otherwise, network operators can continue using the current servers and protocols that already exist on their network architectures. In this work, the SDN controller orchestrates the data plane and only implements passive detection measures. The advantage this approach provides is the SDN controller need only monitor a subset of packets for various flows, which saves compute for other network orchestration/security tasks.

In this work, two controllers are utilized. The first is POX [31]. The second is Ryu [17]. While Pyretic is a valuable framework and serves as the foundation for our work (i.e.,

Network Flow Guard) developed in Chapters 2 and 3, it does have significant limitations. The POX [31] controller, thus the Pyretic [16] programming framework, only allows network operators access to 12 packet header fields for matching, and this proved too much of a limitation for more advanced intrusion detection and prevention methods (e.g., NAT and rogue AP detection). However, Ryu grants access to far more packet header fields, but lacks the modular design capabilities of Pyretic. So, to accommodate the modular nature of the Network Flow Guard architecture developed in Chapters 2 and 3, this work introduces Ryuretic [5] in Chap. 4 as a modular programming framework for Ryu [32]. Its modularity means that programmers can easily create fine-grained, layer-specific, programs (e.g., load balancing, IDS, IPS, traffic engineering, etc.) separately, and then integrate them into the switch via the Ryuretic framework.

In keeping with our goal of minimizing network operator involvement and reducing SDN controller loads, Chap. 5 introduces a Trusted Agent and a communication protocol as part of a transition framework for interacting with the SDN controller. This framework eliminates a significant weakness of it previous chapters–that being the absence of an automated revocation capability for policy enforcements (i.e., move from deny back to allow). Without this capability, network operators must manually reset the controller or update access control lists. Then, in Chap. 6, a new security system is created that capitalized on the benefits of the transition framework where the Trusted Agent becomes an instrument for active testing methods. Moreover, the SDN controller continues to orchestrate network traffic and implement passive security measures.

1.4 Technical Approach

This dissertation essentially progresses in four phases as shown in Fig. 1.1. Throughout these phases, this work transitions from simple detect and deny security applications to a much more robust system for network security and management. These phases and their corresponding chapters are now discussed.

In the first phase (chapters 2 and 3), this work investigates the capabilities of SDN to mitigate existing attack vectors (e.g., rogue DHCP server detection [33] and spoofing of ARP packets [34]) along network edge-devices. We find that these solutions are easily implementable because SDN makes a tremendous amount of state information available to edge-devices. Consequently, this availability of state allows for the rapid development of new, extensible, and inexpensive SDN-based solutions that augment a mac-learning switch to mitigate edge-based network attacks. Moreover, these SDN-based solutions are easily adaptable to current network infrastructures (and protocols) or cloud environments.

The second phase (Chap. 4), recognizes certain limitations of the framework used in the chapters of the first phase. As a result, this phase introduces a programming framework, introduced as Ryuretic [5], for modular, fine-grained, SDN application development. Accordingly, network operators are able to utilize Ryuretic's abstractions to implement more robust network security and management applications for their networks.

In phase three (Chap. 5), the weakness of the previous phases, which required the network operator's continued involvement with network policy configurations, is addressed. It offers a Trusted Agent [30] to help automate simple security policy transitions and further capitalize on the promise of SDN and NFV to reduce network operator workloads. Moreover, by introducing a Trusted Agent, automated network configurations serve to eliminate manual configuration errors relating to policy enforcement transitions.

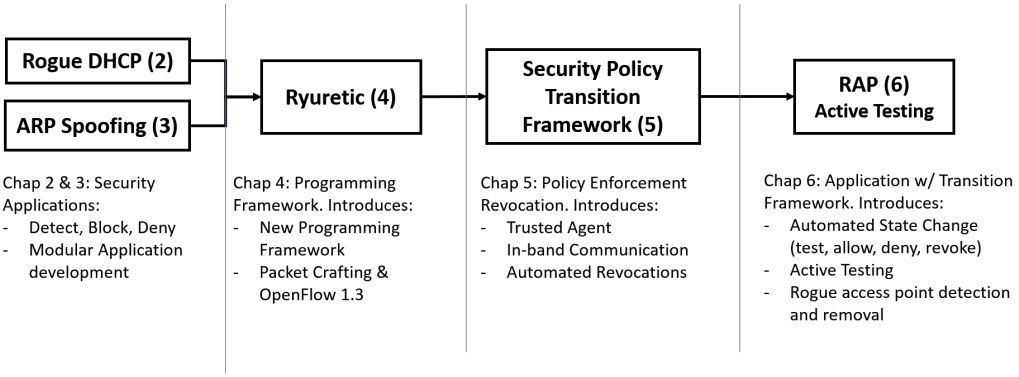


Figure 1.1: Evolution of dissertation concepts

Finally, the fourth phase (Chap. 6), builds off the previous phases to introduce expanded network management capabilities, and it introduces an active measure for detecting malicious activity. These features are made possible by the Ryuretic programming framework and the adoption of a Trusted Agent. Additional details are provided in the following outline and subsequent chapters.

1.4.1 Outline

In the chapters that follow, this work seeks to understand current network operator needs and address them in within the context of a SDN programming framework. As such, each chapter seeks to progress these efforts from addressing established network security issues using SDN capabilities to developing our own programming framework and eventually incorporating additional elements, such as a Trusted Agent, to automate policy transitions and aid active testing.

Each chapter also includes an analysis of an example SDN-based system emulated in Mininet or Mininet-WiFi test environment. These results are encouraging and confirm the effectiveness of our approach to enable network operators to leverage SDN's capabilities for their own applications, whether they be security, traffic engineering, or policy handling. The work offered in this dissertation likewise offers a basis for further expansion of our Ryuretic programming framework, and our Network Flow Guard security applications.

Chapter 2 - Leveraging SDN to Improve the Security of DHCP

Current State of the art technologies for detecting and neutralizing rogue DHCP servers are tediously complex and prone to error. Network operators can spend hours (even days) before realizing that a rogue server is affecting their network. Additionally, once network operators suspect that a rogue server is active on their network, even more hours can be spent finding the server's MAC address and preventing it from affecting other clients. Not only are such methods slow to eliminate rogue servers, they are also likely to affect other

clients as network operators shutdown services while attempting to locate the server. In this chapter, we present Network Flow Guard (NFG), a simple security application that utilizes the software-defined networking (SDN) paradigm of programmable networks to detect and disable rogue servers before they are able to affect network clients. Consequently, the key contributions of NFG are its modular approach and its automated detection/prevention of rogue DHCP servers, which is accomplished with little impact to network architecture, protocols, and network operators.

Chapter 3 - Leveraging SDN for ARP Security

Insider threats are a growing concern for industry, government, and campus networks. Yet, vulnerabilities inherent in Address Resolution Protocol (ARP) are exploitable by insiders seeking to launch sophisticated attacks on local area networks (LANs). Such attacks, initialized through ARP spoofing, include denial of service, server redirect, and man-in-the-middle attacks. Unfortunately, the current state of the art technologies for detecting and preventing ARP poisoning are tediously complex, slow to detect, and difficult to maintain. However, software-defined networking (SDN) enables the implementation of novel security measures that are capable of detecting and eliminating ARP spoofing before it can impact other hosts. Hence, this chapter presents Network Flow Guard for ARP (NFGA), an SDN security module that augments simple, MAC-learning, protocols on OpenFlow-enabled switches. NFG works by hashing a host's physical address with an appropriate port-IP association to deny ARP spoofing at real-time. Moreover, our framework's key contribution is that it achieves ARP security with minimal intervention by network operators while supporting both dynamic and static port allocations, requiring no changes to the network's topology or protocols, and requiring no client software installation.

Chapter 4 - Ryuretic: A Modular Programming Framework for Ryu

We present Ryuretic as a modular, programming framework for SDN application development. Ryuretic draws its inspiration from Pyretic, which already offers powerful, modular abstractions for network operators; however, Ryuretic also builds on Ryu's greater variety of packet match fields and its access to advanced OpenFlow protocols. This framework allows programmers to create new, extensible, and more powerful network applications at a much higher level of abstraction. As Pyretic does with POX, Ryuretic places an additional abstraction layer over the Ryu framework for network application development. Additionally, its abstractions allow researchers to target specific layers of the OSI model and install both proactive and reactive rule sets, without immersing themselves in the Ryu architecture. Ryuretic is under continual development, and it is available to users via GitHub.

Chapter 5 - Security Policy Transition Framework for Software Defined Networks

Controllers for software-defined networks (SDNs) are quickly maturing to offer network operators more intuitive programming frameworks and greater abstractions for network application development. Likewise, many security solutions now exist within SDN environments for detecting and blocking clients who violate network policies. However, many of these solutions stop at triggering the security measure and give little thought to amending it. As a consequence, once the violation is addressed, no clear path exists for reinstating the flagged client beyond having the network operator reset the controller or manually implement a state change via an external command. This presents a burden for the network and its clients and administrators. Hence, we present a security policy transition framework for revoking security measures in an SDN environment once said measures are activated.

Chapter 6 - Leveraging SDN and WebRTC for Rogue Access Point Security

Rogue access points (RAPs) are unauthorized devices connected to a network, providing unauthorized wireless access to one or more clients. Such devices pose significant

risk to organizations, since they provide a convenient means for hackers and insiders to hide malicious or unsanctioned activities on industry, government, and campus networks. Yet, limitations inherent in traditional networks make detecting and removing such devices expensive, time consuming, and difficult to implement. For software-defined networks (SDNs), the risk of a network compromise due to RAPs is equally concerning, and methods for detecting RAPs within SDN architectures are needed. Hence, our work leverages the capabilities of an SDN along with a Trusted Agent (TA) to detect and deny RAPs access to networks by using both generic and novel methods with minimal impact to performance. Three other contributions are included in this work. They include: 1) utilizing an emerging web architecture (webRTC) to detect hidden subnets; 2) developing the first, security-based, use case for Mininet-WiFi, a software-defined wireless network (SDWN) emulator; and 3) enhancing Ryuretic, a modular programming language for SDN application development.

Chapter 7 - Conclusion

This dissertation concludes in Chapter 7 with the following sections: additional results, discussion, contributions, future work, and conclusion. In the additional results section, we compare the abstractions and performance of this work's programming framework, Ryuretic, with that of other established programming languages not discussed in chapters 4 - 6. The discussion section provides a recap of the work conducted in this dissertation, highlighting the evolution of this work from simple detection and mitigation schemes to a complete security policy transition framework with active testing enabled. Next, the contributions of this work are restated, and potential directions for this research are offered as future work. Finally, this chapter finishes with a conclusion.

1.5 Bibliographic notes

Leveraging SDN to Improve the Security of DHCP appears as a short paper in the Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization [33]. Leveraging SDN for ARP Security appears as a full paper in IEEE SoutheastCon 2016 [34]. Ryuretic appears as a full paper in IEEE MilCom2016 [5]. A Security Transition Framework for SDNs appears as a full paper in SNS2016 [30]. An expanded redraft of this publication is also under review as an invited chapter in the book, "Guide to Security in SDN and NFV - Challenges, Opportunities, and Applications," which is scheduled for publication by Springer in 2017. Lastly, Leveraging SDN and WebRTC for Rogue Access Point Security is currently under review with IEEE Transactions on Network and Service Management. In this dissertation, these publications do not appear in their original format. To some extent, material that could not be added to original papers due to space constraints has been included in the chapters that follow. Furthermore, many of these papers have been modified to better segue from one chapter to the next, and all have undergone changes to meet Georgia Tech's formatting guidelines.

CHAPTER 2

LEVERAGING SDN TO IMPROVE THE SECURITY OF DHCP

2.1 Introduction

Dynamic Host Configuration Protocol (DHCP) servers are utilized across campus, industry, and government networks by network operators to provide clients with an Internet Protocol (IP) address, subnet mask, default gateway (DG), and other configuration information. For networks having under-provisioned IP space, DHCP servers are especially important for allowing hosts access to the network, and this requirement grows evermore as organizations seek to embrace the Internet of Things (IoT). Consequently, this combination of under provisioned IP space and the growing use of network devices (requiring access to network services) creates a vulnerability that can be exploited by rogue DHCP servers.

We consider a rogue DHCP server to be any unapproved DHCP server on a network. Sometimes the server appears quite by accident. A user might incorrectly attach a personal router to the network via an open LAN port with default settings that answer DHCP discovery queries with DHCP offers. Accordingly, when such events occur, clients connected to the subnet may receive unsupported IP addresses and lose access to the web and network resources. Of course, the assignment of this unsupported IP address may or may not be mentioned in customer complaints. The malicious case occurs when rogue servers are placed on the network intentionally and for the express purpose of directing victim hosts to the attacker's default gateway. This places network operators in a precarious position of protecting network clients without disrupting their access to the network and its resources. This is not an easy task on traditional networks where network operators must use a command line interface (CLI) to program proprietary systems at the device level.

As we will discuss in Section 2.2, configurations to prevent rogue DHCP servers from

affecting hosts, when possible, are laborious and prone to error. Finding and blocking rogue DHCP servers are likewise tediously challenging and time consuming. Therefore, network operators require a new paradigm for designing and managing networks to remove such threats [20, 6]. Software-defined Networking (SDN) offers such a paradigm and is fully capable of detecting and blocking rogue DHCP servers. Consequently, the key contribution of this work is its modular design and automated detection and elimination of rogue DHCP servers with minimum impact to network architecture, protocols, and network operators.

This chapter is organized as follows. Section 2.2 discusses the current state of the art for rogue DHCP server detection and removal. Known work related to our research is discussed in Section 2.3. In Section 2.4, we provide an overview of our project's design and implementation. We then discuss our test environment and test results in Section 2.5. In Section 2.7, we discuss future work, and finally conclude in Section 2.8.

2.2 State of the Art

Current methods for preventing, detecting, and removing rogue DHCP servers from networks are laboriously time consuming. So much so, network operators can fail to implement necessary configurations correctly or neglect them altogether. We now consider in greater detail the methods available to network operators for detecting and mitigating rogue servers.

Initial methods might include the direct configuration of network switches. For example, provided network switches are capable of DHCP snooping [35], network operators might attempt to configure trust relationships for all switch ports connecting to the DHCP server, transit switches, and hosts. However, with a typical switch possessing 24 to 48 ports, such tasks are laborious for network operators and prone to error since configurations must be completed for each individual port via CLI. Even within a vendor-centric network, switches and routers support a range of features, which are rarely, if ever, universally realized by network operators [36]. If multiple vendor devices are incorporated,

then configurations are further complicated. Even when all configurations are completed correctly, a power surge or future operator error can still occur and cause a traditional network switch to factory reset and remove trust relationships unbeknownst to the network operator. Other proprietary solutions for server management also exist, but they call for vendor lock-in and device registration in an active directory.

Once a rogue DHCP server is suspected, network operators must still employ a variety of methods for locating the rogue server. For instance, diagnosis procedures include (1) disabling the main DHCP server to determine if hosts continue to receive IP addresses from the rogue server (or investigating host log files for a DHCP-server-identifier), (2) obtaining the IP address of the false default gateway, (3) pinging the default gateway to populate the host's ARP table, (4) viewing the ARP table to obtain a mapping of the IP address and the physical MAC address, (5) setting up a continual running ping to confirm when the device is taken down, (6) opening and reviewing the list of MAC addresses contained in the MAC address table of each switch until the offending MAC address is identified, (7) identifying the port hosting the offending MAC, and if found, (8) shutting down the port [37]. Of course, if multiple MACs are associated with the port identified in (7), then this indicates that another switch is hosting the rogue DHCP server and steps (6 and 7) must be repeated until step (8) can finally be completed [38, 37]. Moreover, step (1) indicates that these methods will deny other clients access to the network while operators attempt to determine whether a rogue server is active on their network.

Network operators may also deploy network devices and tools to assist them with detecting rogue servers [38]. For example, network operators may use simple network management protocol (SNMP) to pull MAC addresses and ARP tables from the switch fabric to find the rogue server's MAC address [37]. Network operators may even use sniffing technologies like *Wireshark* [39] or *tcpdump* at various locations on their network to identify the offending server. Of course, network operators may skip the location phase altogether by utilizing a mass network configuration tool to apply new security policies. For instance,

when a rogue DHCP server appeared on the campus network of Georgia Institute of Technology this past year, network operators utilized a Linux-based, multi-vendor tool, Really Awesome New Cisco Config Differ (RANCID) [40], to deploy DHCP snooping across the affected LANS and block the server. However, tools like RANCID still do not automate the process for discovery and removal of rogue servers. As a result, operators are still highly involved, either at the CLI level or scripting level, when detecting and removing such devices from their network.

Regrettably, the above tools and methods represent the state-of-the-art available to network operators. For a proficient network operator receiving an initial report, determining that a rogue DHCP server is on the network may take hours to days depending on the size of the network, location of servers, work priorities, and number of affected users. Additionally, once a conclusion is reached, finding and isolating said server can take one or more hours to complete depending on the network infrastructure [36].

2.3 Related Work

The closest work to our own is a relatively recent publication by Rietz et al. [41], which implements an OpenFlow controller, based on the RYU framework [17], to offer rogue DHCP server protection in virtual environments. Their solution is specifically designed to use the OpenFlow controller to handle all DHCP requests. Thus, it is responsible for generating IP addresses, subnet mask, DG, and other network information for all clients on the network and allocating it to them via DHCP offers. As a result, no other hosts on the LAN are able to detect a DHCP request (or offer) other than the requesting host, hence preventing a rogue DHCP server from detecting or responding to DHCP requests. Regrettably, this solution places an added burden on the controller by requiring it to maintain its own MAC address and IP address tables. It also takes away compute cycles from the controller that are better suited to processing flow requests and updating flow tables, which could affect its scalability. Likewise, the extensibility of their model is in question since future updates

and security additions will require significant changes or overhaul of the controller. Such modifications could also affect switch functionality.

Our model, instead, incorporates its security features via a secondary module, which matches on DHCP flows, and maintains existing DHCP protocols already utilized by traditional networks. This modularity allows the simple switch application to run on the controller (performing its primary function of directing traffic flows) while security is provided by Network Flow Guard (NFG). This modularity further provides extensibility since new security modules can be added as needed without modifying the existing applications. Scalability is also improved as NFG does not respond to DHCP traffic, but utilizes the existing network infrastructure (i.e., the existing DHCP server) to handle DHCP protocols. As a result, no topology changes are required by NFG, save an OpenFlow [42] switch.

2.4 Design and Implementation

In this section, we discuss how an OpenFlow switch, linked to a POX controller running NFG, can replace existing switches. We also highlight how network reconfiguration techniques, enabled by SDN, are utilized to circumvent rogue DHCP servers on a given subnet.

2.4.1 NFG Design

The Network Flow Guard (NFG) design enables existing network protocols so as to minimize its impact to the current network topology. NFG capitalizes upon Internet standards specified in RFC 2131 [43] requiring DHCP to use UDP as its transport protocol and the Bootstrap Protocol (BOOTP) [44] requiring specific source and destination ports be used. For instance, to gain a local IP, clients broadcast a *DHCPDISCOVER* message on their local subnet from source port 68. In turn, DHCP servers respond with a *DHCPOFFER* message, which includes an available IP address, from source port 67. Since multiple servers can respond with a *DHCPOFFER*, the client responds with a *DHCPREQUEST* broadcast, which notifies all other servers that an offer was accepted. By using OpenFlow's header field

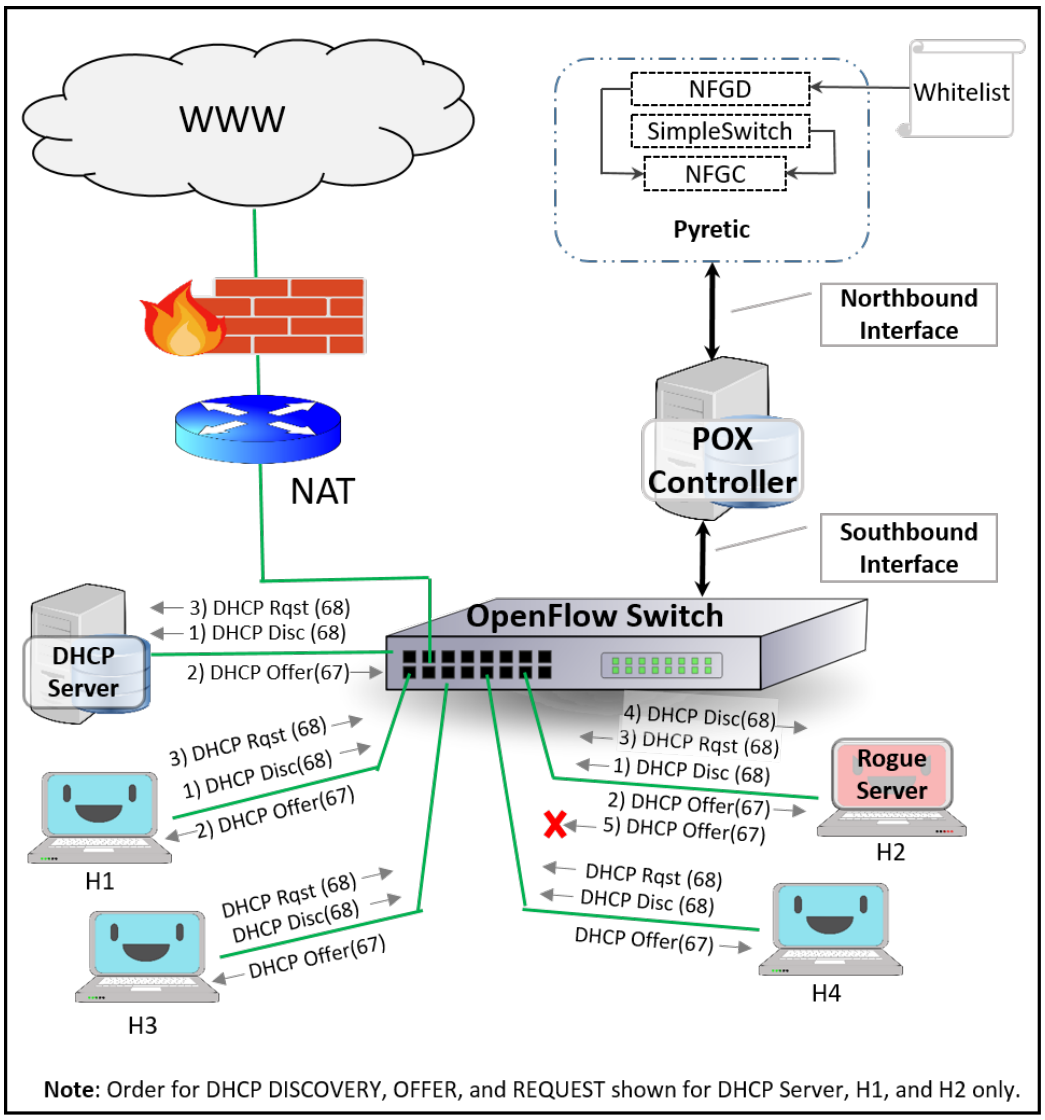


Figure 2.1: NFG Implementation.

match capability, NFG monitors all port 67 traffic, uniquely related to DHCP services, in order to validate that the correct server is issuing IP addresses to clients.

As depicted in Figure 2.1, we utilize the Pyretic [16] framework to develop NFG's Coupler (NFGC) and DHCP (NFGD) modules. The NFGC module couples our NFGD module with Pyretic's native MAC-learning (or simple) switch application. This coupling allows the NFGD module to run concurrently with the switch application and is responsible for filtering suspicious network traffic before it can affect DHCP services on the network. When validating *DHCPOFFERS*, NFGD references a preloaded *whitelist* (maintained by the net-

work operator) of approved DHCP servers. This list can also be updated at the operator's discretion. NFGC then combines the rules from the NFGD module and the MAC-learning switch module and applies them to the POX [31] controller through its Northbound Interface. The POX controller then interprets the instructions and uploads them to the OpenFlow [42] switch via the Southbound Interface.

We choose to use Pyretic for implementing NFG because it is a modular, programming language that allows network operators to implement network applications for SDNs at a much higher level of abstraction [20, 16]. These abstractions allow network operators to develop network applications that provide logically centralized network control as though the SDN controller is handling every packet. For instance, these abstractions allow network operators to target and match on virtual header fields, such as source IP (srcip), destination IP (dstip), source port (srcport), and other metadata. After obtaining specific field matches, Pyretic's action designators specify when packets should be forwarded to the controller for additional processing and when they should be routed to designated output ports.

Accordingly, NFG monitors all srcip and srcport combinations passing through its switch fabric. Packets having DHCP field headers (i.e., having Ethernet type = 2048, protocol = 17 (UDP), and source port = 67) are passed to a DHCP_resolver function, which verifies that the DHCP packet contains a valid *DHCPOFFER*, which is determined by lookup table based on the network operator's provided *whitelist*. NFG then develops policies for arriving DHCP packets. If a DHCP packet is found to be valid, then the corresponding action is set to forward the packet in accordance with the current flow table entries (set by the simple switch application), else the required action is set to drop. The action is then applied to the global forward policy, which is asserted on the OpenFlow switch's flow table. A high-level view of NFG's query and policy implementations are depicted as a flowchart in Figure 2.2.

Predictably, network policies and connections can be highly dynamic and mappings can change frequently–another reason we choose to use Pyretic [16]. Pyretic's Dynam-

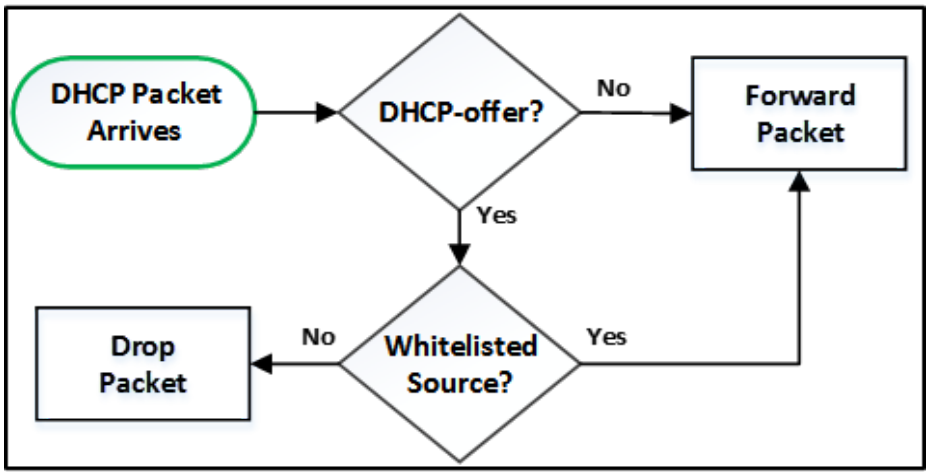


Figure 2.2: NFG Flowchart for DHCP Traffic.

icPolicy class enables policy updates even as the control program is running. This feature allows NFG (via Pyretic) to generate instructions (e.g., forward, encapsulate, or drop) via its Northbound Interface to the POX [31] controller which passes interpreted instructions to our OpenFlow [42] switch through its Southbound Interface as also shown in Figure 2.1.

2.5 Test Environment and Results

Tests were conducted using the Mininet [24] framework. For this environment, we utilize an OpenFlow [42] enabled virtual switch connected to a POX controller [31], an ISC-DHCP server [45], six hosts running a Linux Ubuntu 14.04 OS, and a Network Address Translator (NAT). Note, we only depict four hosts in Figure 2.1 to minimize clutter in our diagram. In this environment, we choose Host 2 (H2) to implement our rogue DHCP server. Within H2, the rogue server is implemented using UDHCPD –a very small DHCP server–to receive *DHCPDISCOVER* packets and respond with *DHCPOFFER* packets [46].

The valid DHCP server is initialized with a static IP address of 10.0.1.200/24. The gateway router is also statically configured to be 10.0.1.200/24. All clients, including the rogue DHCP server, receive their IP address, network mask, and default gateway address via DHCP allocation. Additionally, for the purposes of this experiment, we assign the valid DHCP server an IP address range of 10.0.1.10-30, and the rogue DHCP server an IP range

of 10.0.1.50-60. While the range of IPs could have easily overlapped and encompassed the entire /24 IP space, we implement our experiment this way to quickly identify the legitimacy of IPs found in Wireshark [39]. We then run three separate experiments in our test environment. In the first test, our network uses only a simple switch. In our second test, we initialize our network as before, but with the NFGD module enabled. Our final test begins with a simple switch application, but has the NFGD application enabled after the network is in place and after the rogue DHCP server has affected hosts H3 and H4. Wireshark is utilized from startup for all tests to monitor traffic flows to our OpenFlow switch and our rogue server (H2).

During our first experiment with NFGD disabled, we observe that H1 and H2 are correctly assigned IP addresses from a valid IP pool. Next, after H2 receives a valid IP, it initializes its rogue DHCP server and begins intercepting *DHCPDISCOVER* packets and transmitting *DHCPOFFER* packets. As a result, H3 and H4 are allocated IPs belonging to the rogue DHCP server, and our attacker successfully denies services to these clients. In our second experiment, the NFGD module is enabled. Again, H1 and H2 are correctly assigned IP addresses, and H2 initializes its rogue server, intercepts *DHCPDISCOVER* packets, and responds with *DHCPOFFERS*. However, in this scenario, the NFGD module accurately captures port 67 traffic departing H2 and blocks it, so the requesting host receives a valid *DHCPOFFER* from the authorized DHCP server. The third experiment simply tests NFG's ability to dynamically contribute to network security. To do so, we run the first scenario again with the NFGD module disabled. As expected, we receive the same results. However, this time, without restarting the network, we simply reboot the controller with the NFGD server enabled. We observe that hosts can now simply wait for their current *DHCPOFFER* to expire or issue a *dhclient* command via CLI to obtain a correct IP since the rogue DHCP server is again blocked. Additionally, since state is maintained by the OpenFlow switch, flow tables are unaffected while the controller reboots, so no additional impact to the network is experienced.

2.6 Discussion

As we discussed in Section 2.1, campus, government, and industry networks are susceptible to rogue DHCP servers. In fact, an Internet search for rogue DHCP servers yields numerous forums addressing how to find and remove such devices. Often, the culprit is a student plugging a wireless router into the LAN or a third party organization connecting their network equipment to a hosting company's infrastructure. Georgia Institute of Technology's incident is just one of many frequent occurrences. Yet, such events can also be intentionally malicious, posing far greater concerns and warranting expedited action. Such attacks can DoS the network, steer clients to unintended servers, or allow the attacker to set conditions to implement a man-in-the middle attack.

We also demonstrated in Section 2.5 that the SDN paradigm offers network operators a novel means for denying malicious activity on their networks. Likewise, as previously stated, we believe our new security architecture's key contribution is its innovative use of SDN capabilities to detect and isolate rogue DHCP servers before they have the opportunity to affect other hosts, while requiring no changes to the current network topology, save the incorporation of an OpenFlow switch [47]. Moreover, it significantly reduces the network operator's involvement in the process to simply updating the valid DHCP server *whitelist*.

Perhaps the most exciting fact is that, with no change required of the underlying topology, NFG is a viable alternative for network operators looking to upgrade their network's capabilities. As for network operators who are new to the network (e.g., consultants) and unfamiliar with the location and identity of the valid DHCP servers, obtaining this information is not very difficult. For instance, we developed a program using Scapy [48] to generate a *DHCPDISCOVER* query and then display all *DHCPOFFER* packets received along with their respective server information. Similarly, Wireshark [39] and NMAP [49] queries can be used to identify all active DHCP servers on the network as well. Once all valid DHCP servers are identified, the network operator need only update the DHCP server *whitelist* for

NFG's DHCP module and activate the application in the NFG Coupler module.

We also envision NFG being relevant to other developing architectures, such as Software-defined Exchanges (SDX), Hybrid Networks, and cloud and network functions virtualization (NFV) environments. Small office/home office (SoHo) networks might find this technology helpful as well. Furthermore, we see great potential for NFG's modular nature to incorporate many other network security features, such as denial of service prevention [14], detection and mitigation of port scans [13], intentional network monitoring [20] and anomaly detection [15, 13].

Finally, while we concede that tools exist for detecting, locating, and isolating rogue DHCP servers, there is no tool, currently available, that performs all the above functions. Moreover, none of them do so without significant operator involvement. In contrast, NFG automates this process for the network operator so long as the approved DHCP server information is obtained. Network Operators need only upload their known-good DHCP server information to the NFG's DHCP whitelist and NFG's DHCP module performs the rest.

2.7 Future Work

While our initial findings are promising for implementing security on edge-devices using SDN, other possible research directions of this work might include evaluating the effects of *DHCPOFFER* flood attacks against SDN-based security systems, like NFG. As previously mentioned, this work only considers security measures for traditional attacks. It does not consider actual attacks against the SDN controller. As a result, such future research would be beneficial and complimentary to our own. Additionally, network operators could benefit from research evaluating NFG's DHCP solution against other SDN-based security solutions on physical networks and at larger scale. Doing so, would provide network operators with a better understanding of how such security solutions affect their network's performance and scalability.

2.8 Conclusion

In this work, we explored the complexity of defending against rogue DHCP servers and offer NFG as an SDN solution for detecting and blocking these devices before they affect the network. NFG is scalable because it works with existing DHCP protocols instead of implementing them. It can also be utilized within established network infrastructure simply by replacing edge-devices with OpenFlow switches. NFG is extensible in that future modules for denial of service attacks, port scans, ARP poisoning, anomaly detection, and other threats can easily be implemented as independent modules. Still, despite our confidence of SDN's role in network security, it is likely that features such as ours may still need to be incorporated as part of a defense-in-depth strategy that still requires middleboxes at various tiers within campus, industry, and government network architectures. Even so, we believe that NFG offers an excellent opportunity to enhance network security at the edge for numerous, developing network topologies, including SDX, enterprise networks, hybrid networks, and cloud and NFV infrastructures.

CHAPTER 3
LEVERAGING SDN FOR ARP SECURITY

3.1 Introduction

Hackers continuously test the limits of tools available to network operators, making network security a growing concern. While address resolution protocol (ARP) aids network devices to locate the physical address of a provided Internet Protocol (IP) address, ARP is also extremely vulnerable to attack, serving as an attack vector for hackers seeking to masquerade their media access control (MAC) address as belonging to a legitimate IP address. Insiders are able to exploit ARP to masquerade their MAC address as belonging to other legitimate IP addresses by spoofing the ARP packets they send to victim hosts. To do so, attackers take advantage of the trust afforded ARP-replies by spoofing ARP packets destined for the victim host. Doing so causes the victim to associate a legitimate IP address with the attacker's MAC address. With this association, the attacker is able to conduct denial of service (DoS) attacks, man-in-the-middle (MITM) attacks, or server redirection attacks. Although various methods and tools present counter measures to ARP poisoning (e.g., static configurations, host software, and proprietary devices), few of these options offer an ideal solution. As we will discuss in Section 3.3, current solutions are challenged to address scalability, maintenance, cost, vendor neutrality, operating system (OS) compatibility, bring your own device (BYOD) initiatives, and power and space constraints. Likewise, traditional networks, with features like dynamic ARP inspection (DAI), require that network operators complete configurations on each individual switch in their network.

Instead, we propose to use software-defined networking (SDN), a new paradigm that decouples the control plane from the switch's data plane, to provide security and defend against ARP spoofing attacks. In this chapter, we present Network Flow Guard as a mod-

ular application that augments an SDN controller to detect and prevent ARP-replies from unauthorized hosts. Our method does not require any network topology changes (save an OpenFlow-enabled switch), authentication services, cryptographic keys, or significant network operator support. Instead, NFG monitors dynamic host configuration protocol (DHCP) *offers, requests, and acknowledgments* from valid DHCP servers to construct a dynamic table consisting of MAC-IP-port-fixed-state associations for each device on a given LAN. Doing so allows NFG to leverage the capabilities of SDN to thwart ARP poisoning attempts as they occur. Moreover, NFG's key contribution is that it provides ARP security while requiring little network operator intervention, no additional equipment or host software, and no changes to the network's current topology or protocols.

This chapter is organized as follows. Section 3.2 provides background information concerning address resolution protocol and motivates some reasons for securing it. Section 3.3 discusses the current state of the art for detecting and preventing ARP spoofing and ARP poisoning. In Section 3.4, we first provide an overview of NFG's design and implementation. We then discuss our test environment, our custom ARP poisoning scripts, and our test results in Section 3.5. Related work is then discussed in Section 3.6. In Section 3.7 we offer further discussion on the SDN paradigm and its viability towards security. Then, in Section 3.8, we discuss future work, and finally conclude in Section 3.9.

3.2 Background

Switches operate at the layer 2 (i.e., the data layer) of the OSI model and handle communication between physical addresses. These MAC addresses are normally assigned by device manufactures to serve as the device's local address on a subnet. Even in cases where Host A is on a different subnet than Host B, Host A will still use the MAC address of its gateway router to establish a path between itself and Host B. However, this last action often requires Host A to know the layer 3, IP address of Host B.

Assuming that Host A already has Host B's IP address, Host A will utilize address

resolution protocol (ARP) to obtain Host B's MAC address or the MAC address of the device offering access to Host B, such as an intermediate router. This method of converting an IP address to a MAC address is defined in RFC 826 and in IETF international standard, STD37 [50]. Once Host A has Host B's MAC address, Host A caches the IP and MAC address in its ARP table. In actuality, the host generally checks its ARP table or MAC-IP associations first, since previous associations are temporarily cached there. This is done to reduce network traffic, however, in our above example, we assume the MAC-IP association was not found. Additionally, the *ARP-request* is sent as a broadcast message with a MAC address destination of *FF:FF:FF:FF:FF:FF*, causing it to be received by all hosts on the subnet. Accordingly, the host on the subnet owning the requested IP address replies to the ARP message with an *ARP-reply* containing its MAC-IP information allowing Host A to update its ARP table and retain the information until it times out. Normally, machines update their ARP table after receiving an ARP *is-at* reply in response to an ARP *who-has* request, as depicted in Fig. 3.1. However, ARP can also serve as a simple announcement protocol. For example, if Host A moves to a different IP address, it may utilize a gratuitous ARP to have other hosts update their ARP tables. For example, a user can enter the Linux command, *arping A I eth0 10.0.1.200*, and send a broadcast (i.e., *ARP-reply*) from that host to all other hosts on the LAN to have them associate its MAC address with a provided IP address [51]. Since ARP tries to reduce network traffic, it uses all available information from arriving ARP packets to update its table.

Considering that ARP does not use authentication methods for ensuring valid ARP responses on subnets, other hosts can generate replies containing spoofed data and cause other hosts on the network to update their ARP table with fraudulent MAC-IP associations.

#	Time	Source	Destination	Info
1	0.00000	6e:8f:c6:1d:ac:2b	Broadcast	Who has 10.0.1.10? Tell 10.0.1.13
2	0.00017	72:3c:14:e2:74:0a	6e:8f:c6:1d:ac:2b	10.0.1.10 is at 72:3c:14:e2:74:0a

Figure 3.1: ARP Request/Reply Packet Sequence

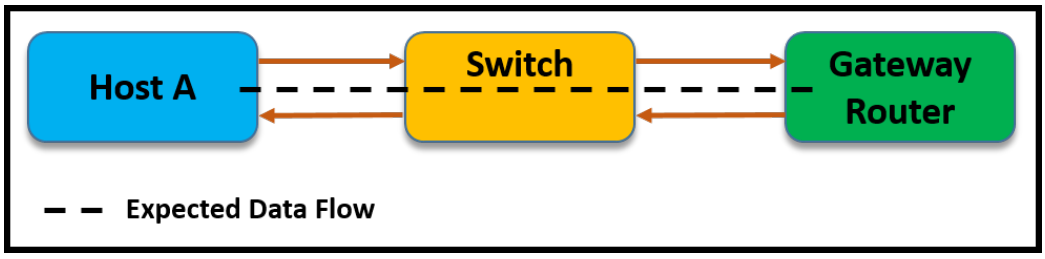


Figure 3.2: Normal network flow.

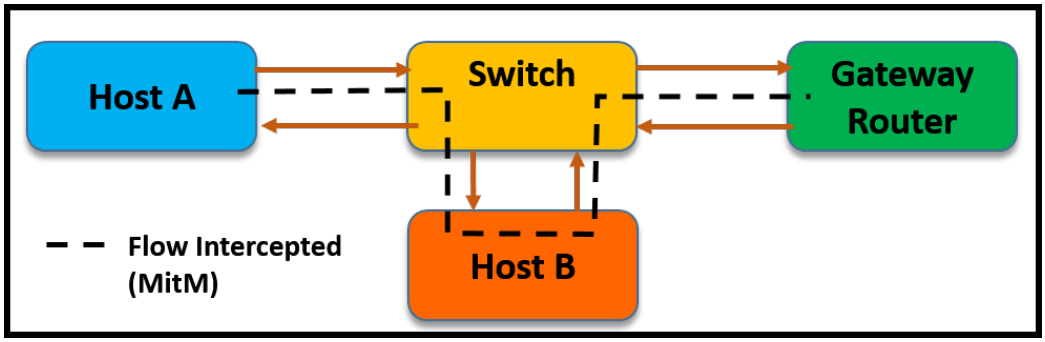


Figure 3.3: Man-in-the-middle.

Additionally, ARP cache poisoning allows insiders to conduct various attacks (e.g., man-in-the-middle, denial of service (DoS), or server redirect). We illustrate these attacks in the figures that follow. Moreover, in Section 3.5, we implement these attacks using custom scripts and validate their effectiveness on LANs using an unprotected MAC-learning switch. We then demonstrate NFG's ability to detect and prevent such attacks. Fig. 3.2 shows the expected flow of data for Host A. However, through ARP poisoning Host B is able to insert itself into the data path. Fig. 3.3 depicts the data flow when a host effectively poisons Host A's cache to associate the attacker's MAC address with the Gateway

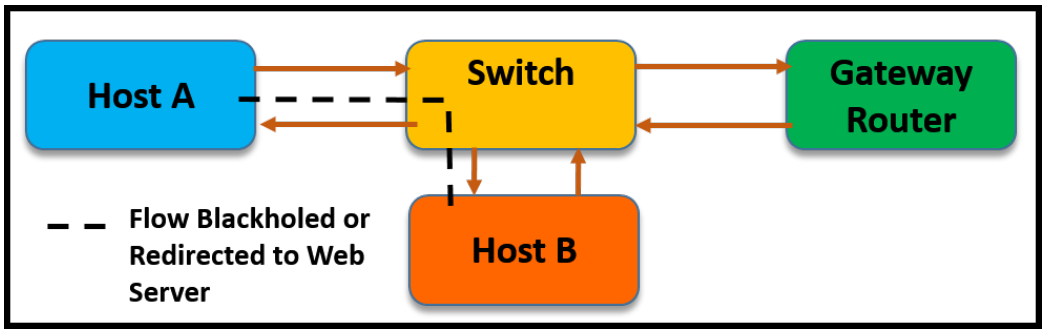


Figure 3.4: Denial of Service (DOS) or Server Redirect

Router's IP. The attacker now receives traffic from Host A, inspects and modifies it, maintains the distant end connection, and logs information of interest like user names and clear text passwords, and other data. The attacker may also choose to simply drop packets that are received in order to DoS Host A or steer Host A to a malicious server as depicted in Fig.3.4.

3.3 State Of The Art

Each host on a subnet maintains its own ARP table (or cache) for mapping IP addresses (layer 3) to MAC addresses (layer 2). Unless statically configured, modern network protocols maintain this table by monitoring ARP packets. As shown in Fig. 3.1, examples of these packets include *ARP-requests* and ARP-replies. Unfortunately, ARP is a stateless protocol [52]. Since hosts do not track the *ARP-requests* they send out or the replies they receive, they will automatically accept responses without having ever sent a request. Additionally, if these ARP packets are forged (or spoofed), receiving hosts are unable to distinguish modified packets from legitimate ARP responses. Host devices then overwrite correct MAC-IP associations with fraudulent ones. The result being that affected hosts forward their packets to the attacker's MAC address. ARP's stateless nature and the fact that it requires hosts to use MAC-IP associations already cached in their ARP table makes ARP poisoning particularly effective against LANs.

Several tools, like Ettercap [53] and Dsniff [54], are available to automate the exploitation of ARP's vulnerabilities. A common attack is to mislead a victim into believing the physical address of the network's default gateway belongs to the attacker. As a result, the attacking device can set itself up as a blackhole, a rogue server, or a proxy (or man-in-the-middle). As countermeasures to ARP attacks, several defense mechanisms have been developed. They include host/server software, static configurations, and proprietary systems. One software solution is Arpwatch [56]. It tracks MAC-IP associations and syslog activities and then emails reports containing changes. It also requires that local Ether-

Table 3.1: ARP Detection and Protection Tools [55]

Product	OS	Protection	Active/Passive
Antidote	Linux	No	Passive
Arp_Antidote	Linux	No	Passive
Arp_Alert	Linux	No	Passive
ArpON	Linux, OSX, BSD, Solaris	Yes	Active
ArpStar	Linux	Yes	Passive
Arpwatch	Linux	No	Passive
ArpWatchNG	Linux	No	Passive
remarp	Linux	No	Passive
Snort	Window, Linux	No	Passive
Winarpwatch	Windows	No	Passive
Xarp	Windows, Linux	Yes Pro-Version	Active+ Passive

net interfaces be enabled for *pcap* listening [56]. Arpwatch can be particularly affective on networks connected via hub (since one machine can monitor all traffic; however, the unicast nature of ARP responses cause Arpwatch to miss many ARP response packets on switched (MAC-learning) networks [52]. If a network operator is to effectively deploy Arpwatch, then nearly every host must have Arpwatch installed. However, network operators may have little control over all network devices due to bring your own device (BYOD) initiatives, proprietary systems, etc. A similar problem arises with software compatibility for various host operating systems on the network, and detecting anomalies on numerous syslog files from each device is also challenging. These constraints make Arpwatch an insufficient option for network ARP protection.

Two other available tools of note are ArpOn [57] and xArp [58]. They too must be installed on all hosts. ArpOn is open source and available for Linux, OSX, BSD, and Solaris. XArp has a free and paid-for version (XArp-Pro) and supports both Linux and Windows operating systems. Both tools, ArpOn and Xarp-Pro, are able to provide detection of ARP poisoning and protection. However, as with Arpwatch, these tools must also be installed on all network devices to be effective, which is not a likely possibility in current network infrastructures. Table 3.1 includes a list of other popular tools for detecting (or protecting) against ARP poisoning with similar limitations [55].

Still, some high-end (i.e., expensive and proprietary) switches allow network operators to declare a single port as a monitor port. Afterwards, the network operator can configure that port to have visibility over all other ports and then install a server to run an ARP poison detection tool (e.g., Arpwatch) from that port for network monitoring. However, this method removes one port from network use, calls for extra equipment (the Arpwatch server is essentially a middle box), requires network operators to make manual configurations to the switch, maintains vendor lock-in, and requires other resources, like power and rack space.

Another option available to network operators is static ARP table configuration. Static configurations cause the kernel to ignore all ARP-replies for IP addresses that are not statically stored in its ARP table [52]. However, this type of configuration requires that each device on the subnet be entered into the ARP table of every device on the subnet that is capable of storing static entries. While static entries can be effective and most versions of ARP allow for a pre-created file to be loaded into each system [52], network operators are challenged to continually distribute address mappings to each host on every subnet they manage. As with other methods, static ARP tables are not scalable, and they present a huge burden for network operators. Additionally, many networks, having a limited range of available IPs, require some form of dynamic host configuration protocol (DHCP) to dynamically add new network devices to subnets. In these networks, static ARP tables are simply too burdensome, and there are not enough IP addresses to ensure that only one is assigned to each MAC address.

Since port-security, static configuration, access-lists, and other well-known security features are not adequate countermeasures for ARP poisoning, Dynamic Arp Inspection (DAI) was developed for some proprietary switches. Its default setting is that all ports are untrustworthy and inspects ARP packets from untrusted ports. As a result, all ARP traffic from untrusted ports are compared against a DHCP snooping database or against ARP access-lists to ensure valid MAC-IP table bindings [59, 60]. However, this protocol

must be enabled per switch and VLAN via CLI by network operators, and it is subject to configuration problems. It is also a proprietary solution used by commercial vendors which further ensures vendor lock-in. Nor can it be extended to SDNs in a controller agnostic way.

3.4 Design and Implementation

In this section, we discuss the goals, design, and components for our Network Flow Guard (NFG) implementation. We first show how an OpenFlow switch, linked to a POX controller using NFG, can simply replace existing switches and provide an immediate impact to security. We then highlight how our design utilizes network reconfiguration techniques, enabled by SDN, to circumvent ARP poisoning within a local area network.

3.4.1 NFG Design

Network operators require solutions that are accurate, extensible, and scalable. Otherwise, these solutions have limited viability for network requirements. Hence, we adhere to the recommendations presented by Song et al. [61] for deploying countermeasures to ARP poisoning:

1. It must control the management costs of hosts.
2. It should minimize cryptographic processing.
3. It should provide timely detection and prevention.
4. It should be easily adaptable to current networks.
5. It should minimize hardware costs.
6. It must be compatible with ARP.
7. It should not slow down the ARP request/reply process.
8. It should consider all ARP attacks
9. Traffic should be contained to the network.

Adhering to these guidelines, NFG is implemented with little impact to the existing network infrastructure. It has no cryptographic requirements, and it can detect and prevent ARP spoofing attempts as they occur. NFG's only requirement is that it have an OpenFlow [42] switch. As a result, it requires no topology changes or protocol changes for its implementation. It also utilizes traditional DHCP and ARP protocols to implement its dynamic table and validate ARP packets. By utilizing the above protocols and only augmenting the MAC-learning switch, NFG is compatible with other protocols and is easily adaptable to other networks. Furthermore, by using host MAC addresses as primary keys for its dynamic table, NFG eliminates the possibility of multiple, identical MAC addresses existing on the same subnet.

From the high-level view, NFG implements its security features by analyzing DHCP and ARP packets, building its dynamic table of MAC-IP-port-fixed-state associations, and matching incoming ARP-replies against the appropriate table entry. In Fig. 3.5, each arriving packet is directed to the appropriate module based on its classification (i.e., ARP, DHCP, or other). Of these modules, only the ARP Validator is able to drop packets. The Dynamic Table Updater simply extracts the information needed to update the dynamic table, which in-turn provides the ARP Validator with verified MAC-IP-port associations, before forwarding the packet.

Since networks often contain a combination of both dynamic and static addresses, NFG

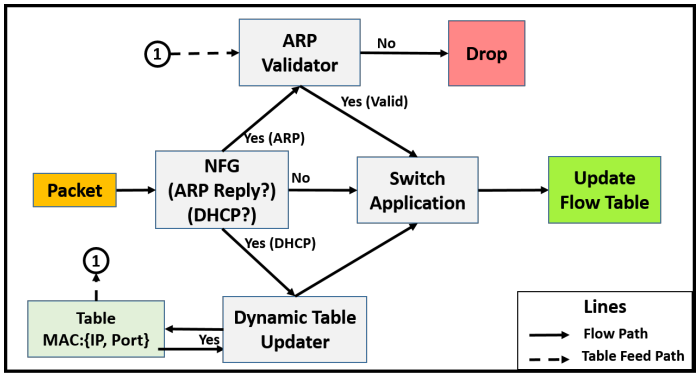


Figure 3.5: High level view of NFG Operation

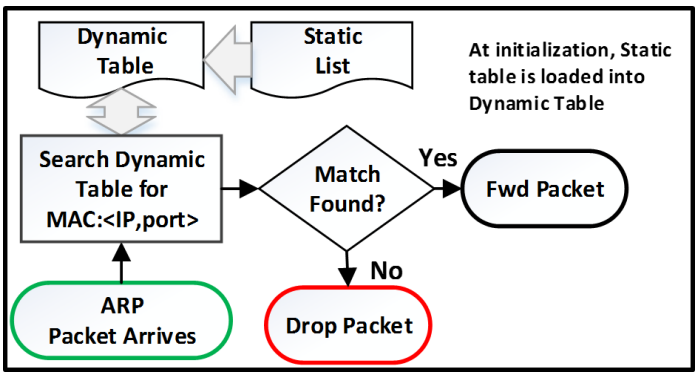
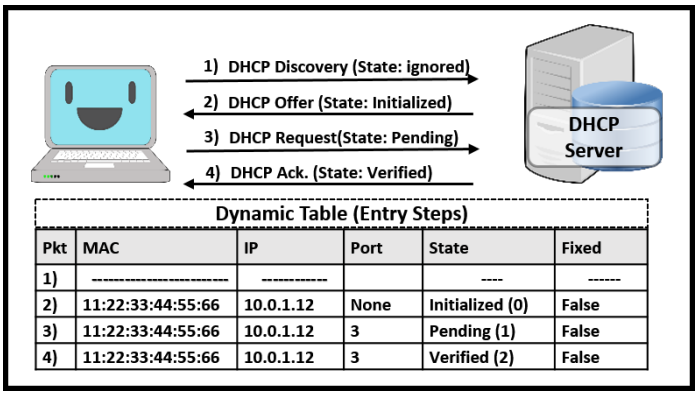


Figure 3.6: NFG ARP Validation Flow Graph



Dynamic Table (Entry Steps)

Pkt	MAC	IP	Port	State	Fixed
1)	----------	----------		----	------
2)	11:22:33:44:55:66	10.0.1.12	None	Initialized (0)	False
3)	11:22:33:44:55:66	10.0.1.12	3	Pending (1)	False
4)	11:22:33:44:55:66	10.0.1.12	3	Verified (2)	False

Figure 3.7: Dynamic Table Entry Steps

supports both static and dynamic port allocations simultaneously. This is accomplished by allowing network operators to submit static MAC-IP-port allocations via a static-list file as indicated in Fig. 3.6. This list is then added to NFG's dynamic table at boot and protected from overwriting via a *fixed field* permission check. The remaining entries are then added via the Dynamic Table Updater module, and the table is used by the ARP Validator to determine the validity of *ARP-reply* packets.

While building the dynamic table, NFG utilizes the MAC address of each network device as its primary key in the table entry. The row for each entry contains an IP address, port number, entry state, and static (or fixed port) indication that is specific to its MAC address. For security purposes, entries to the dynamic table can only be initialized by a valid *DHCP-offer*. Only then can a host's *DHCP-request* be used to assign a port to an established MAC-IP entry and move the entry's state to pending. Once the DHCP server

responds with a *DHCP-ack*, the entry is verified, and that MAC-IP-port association is then allowed to submit ARP-replies. Otherwise, the ARP packet is dropped at the switch. Additionally, as shown in Fig. 3.7, NFG limits interactions with untrusted devices by only using *DHCP-request* packets that are bounded by *DHCP-offer* and *DHCP-ack* packets. *DHCP-discovery* packets are ignored.

The algorithm for handling these DHCP packets and updating the dynamic table is depicted as a flowchart in Fig. 3.8. The updater first checks to see if arriving DHCP traffic is

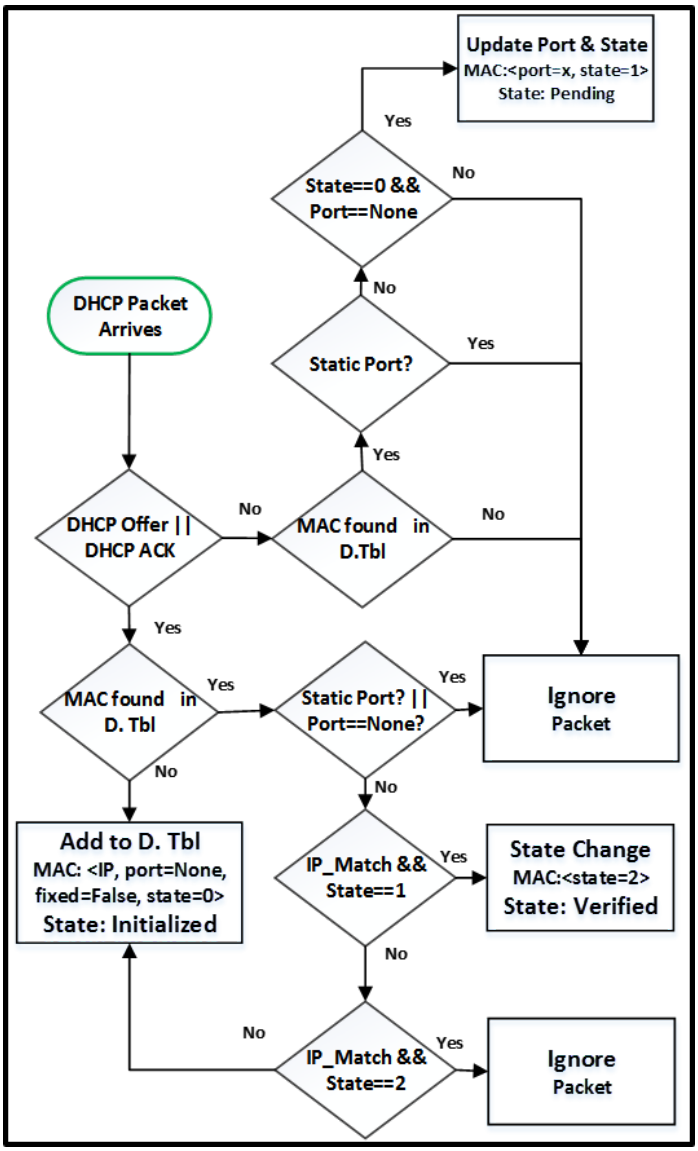


Figure 3.8: DHCP Packet Flow Graph for Dynamic Table

from the DHCP server. If so, the module then confirms whether or not the packet's destination MAC (dstmac) is in the table. If not, then the dstmac and the destination IP (dstip) are added; the port entry is set to None; fixed is set to False; and state is set to *initialized-0*. Otherwise, packets from the DHCP server are either ignored, used to update MAC-IP associations, or used to move the table entry's state to *verified-2*. If the packet is determined to instead originate from the host, then its source MAC (srcmac) is checked against the table. If the MAC address exists and input port is not static, then the table's current port and state are checked. If the entry's state is *initialized-0* and the port is unassigned, then the srcport is assigned to the table entry, and its state is upgraded to *pending-1*. Otherwise, the packet is ignored.

3.4.2 Implementation

As depicted in Fig. 3.9, our implementation utilizes the NFG Coupler (NFGC) module to couple the NFG ARP (NFGA) application with a standard, MAC-learning, switch application. This switch is already a part of the Pyretic framework [16] used for our design. The

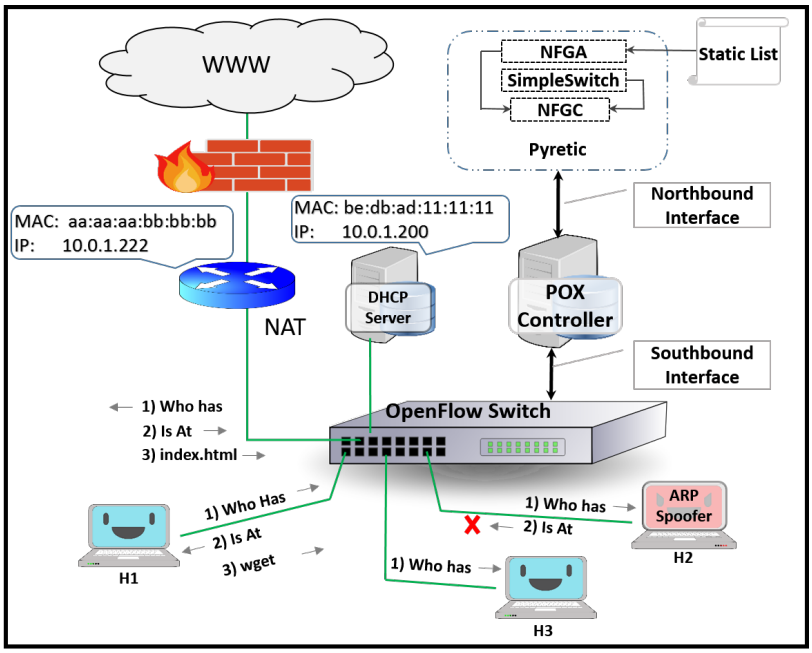


Figure 3.9: NFG Implementation

```
# Capture all ARP traffic.
self.arp_pkts = packets(None, group_by=['srcip','srcport'])
self.arp_pkts.register_callback(self.arp_resolver)
self.arp_in = match(ethtype=2054, protocol=2) >> self.arp_pkts
self.policy = self.forward + self.dhcp_ld + self.arp_in
```

Figure 3.10: Pyretic Query used by NFG

NFG coupler then allows NFGA to generate rules concurrently with the switch application. These rules are then combined by NFGC and pushed through the Northbound Interface to the POX controller, which interprets the instructions and uploads them to the OpenFlow switch via the Southbound Interface.

While OpenFlow switches use three fields (i.e., packet header, action designator, and statistics) to enable matching on any portion of a packet's header and not just the dstip and dstmac, these 'low-end' commodity switches possess limited memory and lack their own dedicated control plane [42]. Fortunately, numerous SDN controllers exist, and we chose the POX [31] controller for our implementation. Our NFG modules, however, are written in Pyretic [16], which is a modular programming language that allows network operators to implement applications for SDNs at a much higher level of abstraction. These abstractions allow for NFG (and other network applications) to provide logically centralized control as though every packet is being handled by the SDN controller. In so doing, virtual header fields like source IP (srcip), destination IP (dstip), source port (srcport), destination port (dstport), and other metadata can be matched to specific data types (e.g., DHCP and ARP). When matches occur, they map to specified action designators that specify whether packets are forwarded to the controller, forwarded to a specified outport, dropped at the switch, or modified. A query can also be utilized to have specific packets forwarded to the SDN controller, and we use this feature to target all DHCP and *ARP-reply* traffic. An example code snippet for ARP-packet queries is provided in Fig. 3.10.

In the first line of Fig. 3.10, we begin a query policy that directs the OpenFlow switch to monitor all srcip and srcport combinations passing through its switch fabric. The second

line provides instructions for handling received packets. Line three provides additional filters, so only *ARP-reply* packets (identified by ethtype=2054 and protocol=2) are received by the callback function in line two. Finally, these rules are added to Pyretic's policy and pushed through its Northbound Interface to the POX [31] controller where it then updates the switch's flow table though its Southbound (OpenFlow) Interface as shown in Fig. 3.9. Predictably, network policies and connections can be highly dynamic and mappings can change frequently–this is another reason we choose to use Pyretic [16]. Its DynamicPolicy class enables policy updates even as the control program is running.

3.5 Test Environment and Results

To test our solution, we developed a test environment within a Mininet [24] framework. Within this environment, we utilize an OpenFlow [42] enabled virtual switch connected to a POX controller [31], a lighttpd [62] web server, a DHCP server, six hosts running a Linux Ubuntu 14.04 OS, and a network address translator (NAT) serving as an intermediate router. Note, we only depict three hosts in Fig. 3.9 to minimize clutter in our diagram. In this environment, we choose Host 2 (H2) to be our ARP spoofer/attacker. Within H2, the lighttpd web server is running. Additionally, we developed a custom ARP attack tool (henceforth referred to as ArpPoison) that utilizes NMAP [63], arpspoof [64], sslstrip [65], and iptable manipulation to implement a blackhole attack, web server redirect, and man-in-the-middle attack.

Using the testbed, described above, the valid DHCP server is initialized with a static IP address of 10.0.1.200/24. The gateway router is also statically configured to be 10.0.1.200/24. These static IPs are recorded in the static-list file. All clients, including the one hosting the rogue host, receive their IP address, network mask, and gateway IP via DHCP allocation. Wireshark [39] and tcpdump are also run to monitor network traffic and verify if ARP-replies reach their intended victim.

To test our design, we run two separate experiments in our test environment. For the

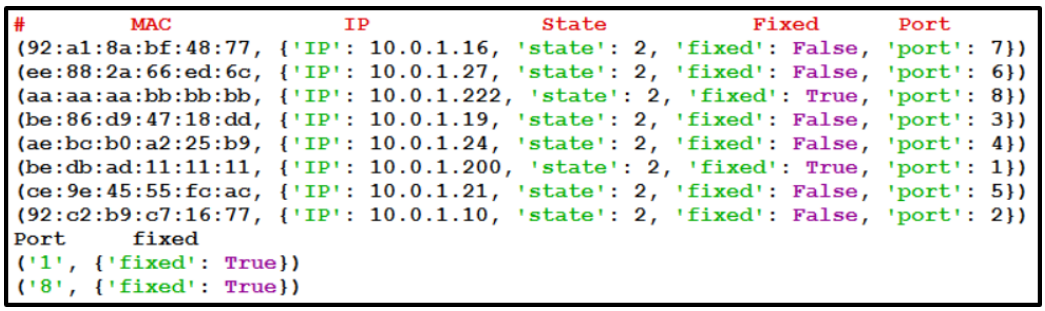


Figure 3.11: Dynamic Table

first test, we run our network using only a simple switch. In our second test, we initialize our network as before, but with the NFGA module enabled. During startup for all tests, Wireshark is used to monitor traffic flows through the OpenFlow switch (s1), the victim host (H1), and the rogue host (H2).

During the first experiment with NFG disabled, we observe that after H2 receives its IP address and ArpPoison is run that all attacks are successful. Not only does the attack blackhole the victim, it successfully steers all web traffic to its rogue web server, and it captures username and password information by running as a man-in-the-middle while a user attempts to access a social media account.

During the second experiment with NFG enabled, we observe that the dynamic table successfully maps all MAC-IP-port-fixed-state associations as depicted in Fig. 3.11. Moreover, while legitimate ARP-replies are permitted, spoofed ARP traffic is successfully detected and dropped at the port after NFG detects the first spoofed ARP packet. The result is that one spoofed packet reaches the victim, but all subsequent packets are denied. Consequently, the victim's ARP table quickly drops the association, and the threat is contained since the attacker is unable to maintain the flow of spoofed data packets to the victim. Therefore, we conclude that NFGA successfully prevents potential blackhole, server redirect, and man-in-the-middle attacks by detecting and dropping spoofed ARP packets at the port.

3.6 Related Work

While current tools and hardware for addressing ARP attacks are discussed in Section 3.3, it is worth reemphasizing that these solutions lack scalability, require significant effort from network operators to configure and maintain, and introduce other requirements like space and power. Unfortunately, the coupling of the control plane and data plane within traditional network devices makes it difficult for network operators to deploy universal solutions and research continues.

S-ARP [66] and T-ARP [67] represent research aimed at table server synchronization methods. However, these schemes suffer from network compatibility issues with configurations, administrative overhead, and protocols [61]. Nam et al. [68] developed a voting scheme that relies on neighboring hosts, yet it depends on fair voting and requires infrastructure and operating system modifications. DS-ARP [61] offers a detection scheme using routing trace to determine whether network paths have moved. It also requires that an agent be installed on hosts and that a server be installed on the network to periodically perform surveillance on ARP cache tables and report suspected ARP spoofing attacks to the server. A routing trace is performed every time an ARP cache table is updated. Of the traditional network solutions, the work completed by Philip [69] most closely resembles our own. In his research, the firmware of a wireless access point (WAP) is upgraded to form MAC-IP associations within a wireless subnet. However, this solution does not account for static configurations required for wired networks (e.g., printers and servers). Neither is it easily implemented on existing networks.

In contrast, a software-defined network (SDN) separates the control and data planes, so network intelligence and state are abstracted away and maintained by a logically centralized controller [9]. As a result, network operators are now able to work with a network operating system (NOS) instead of its underlying infrastructure. Using this paradigm, researchers are attempting to leverage OpenFlow [42] switches to handle various security threats. Hong

et al. [11] observed that SDN is susceptible to a new form of poisoning similar to ARP spoofing attacks on legacy networks and propose TopoGuard as a countermeasure. Yet, they only address Host Location Hijacking attacks and Link Fabrication attacks by preventing the SDN controller from receiving spoofed packets. As such, their work augments NFG since it provides an extra layer of security for the SDN controller, while NFG focuses on the actual ARP packets being sent to neighboring hosts. Sphinx [70] is a similar effort to provide protection for enterprise SDNs by gleaning state and forwarding metadata from OpenFlow control messages. Additionally, it extracts metadata from packets to build a flow graph of MAC-IP bindings and a list of possible switch-ports. However, this appears to be a secondary function of Sphinx, and it only flags possible ARP spoofing attempts while NFG detects and blocks them. Also, since NFG uses DHCP to initiate table entries, only one IP-port association can exist for a MAC address at any given time. Work by Kang et al. [71] seeks to leverage SDN for cloud environments by capturing MAC-IP associations as instances are created in the network. However, their work does not easily translate to a physical network having both static and dynamic port allocations, but NFG could be easily adapted to a cloud environment. Another SDN model utilizes an Enhanced Spoof Detection Engine (E-SDE) [72] to detect ARP spoofing, yet, it does not prevent ARP attacks. Moreover, it generates additional traffic to confirm the legitimacy of ARP packets.

In contrast to the above examples, NFG requires no changes to the network's topology or any additional protocols, and it maintains a MAC-IP-port-fixed-state association for every physical address on the network. Hence, the MAC-learning switch is able to focus on its primary task of directing traffic flows while our security module implements desired security features. NFGC simply couples the NFGA and MAC-learning switch rules together. Furthermore, by using this methodology, we are able to maintain the same base network and ARP protocols already utilized on traditional networks. So *ARP-requests* and *ARP-replies* all function appropriately with no modifications. This approach provides for a less expensive, more scalable, and more extensible solution for detecting and mitigating ARP

spoofing on a local area network.

3.7 Discussion

As we discussed in Section 3.1, most subnets (whether they exist on campus, government, or industry networks) are susceptible to ARP spoofing attacks. These attacks can lead to data breaches, data modification, loss of resources, communication interference, and much more [71]. Accordingly, we developed our own custom scripts to exploit this vulnerability on a standard MAC-learning switch. Yet, by using an existing protocol, like DHCP, in conjunction with the capabilities of SDN, we are able to construct a dynamic table capable of supporting both static and dynamic port allocations. Consequently, our decision to utilize DHCP protocols extends our earlier work offering security against rogue DHCP servers [73].

Ideally, we would have chosen to construct our dynamic table purely from the network operator's static list and packets originating from the DHCP server. However, *DHCP-offers* and *DHCP-acks* only provide MAC and IP address associations–they do not provide a destination port. Additionally, while this information could have been obtained from modifying the switch application to update our dynamic table, it violates our goal of creating security features that act autonomously from the network's current architecture or protocols. As a result, we chose to include *DHCP-requests* (which arrive from hosts) as part of a sequence of events that allow a table entry to systematically move from an initialized state to a verified state.

For our table, we also realized early in our development that MAC addresses would have to serve as our primary key since they provide a one-to-one relationship with IP-port associations. We initially considered using ports as our primary key since we also allow for static port allocations. This configuration works if all ports map to a single MAC. However, ports can have many associated MAC and IP addresses (e.g., virtual machines that are bridged to the network). Likewise, most hosts do not have an IP address when

they join the network, only a MAC and port, so MAC address proved to be our best option. Additionally, we chose to implement a separate table purely for static port verification. Hence, our design allows RFC826 [50] protocols to operate normally without interruption while determining if ARP-replies are legitimate via table lookup.

We also demonstrated in Section 3.5 that the SDN paradigm offers network operators a novel means for denying malicious ARP activity. In contrast, traditional network security features take advantage of the control plane being juxtaposed with the data plane on legacy equipment. Thus legacy security measures are not immediately applicable to software-defined networks where the control plane is separated from its network devices. This difference in paradigms means that new security features are required to address old problems like ARP poisoning. SDN switches are far too simple to implement the complex security protocols offered by legacy network devices. Hence, our framework's key contribution is its innovative use of SDN capabilities to detect and isolate ARP spoofing before it can affect other hosts, while requiring no changes to the current network's topology or protocols, and with minimal network operator requirements. These benefits make NFG a viable alternative for network operators looking to upgrade their network's capabilities.

Finally, while tools and methods for detecting and preventing ARP poisoning exist, there is no tool, currently available, that is both scalable and easily maintained. They either require software deployments or OS upgrades on all hosts, require additional equipment or firmware upgrades for support, or require a significant amount of effort by the network operator to configure and maintain. NFG, however, detects and prevents ARP attacks by building its own table for ARP verification, and accepts a static-list file containing statically allocated addresses.

3.8 Future Work

We believe our initial findings utilizing a Pyretic [16] SDN controller in a Mininet [24] environment shows promise for future secure SDN deployments. We envision NFG being

relevant to other developing architectures, such as software-defined exchanges (SDX), hybrid networks, and cloud and network function virtualization (NFV) environments. Small office/home office (SoHo) networks might find this technology helpful as well. Furthermore, we see great potential for NFG's modular nature to incorporate many other network security features, such as denial of service prevention [14], detection and mitigation of port scans [13], intentional network monitoring [20] and anomaly detection [15, 13].

3.9 Conclusion

In this work, we explored the difficulty of defending against insiders attempting to poison the ARP tables of other hosts on the same subnet. Part of this work included developing our own suite of attack tools (ArpPoison) to demonstrate the effectiveness of these attacks. We also explained how Network Flow Guard uses the field matching capabilities of SDN to detect and drop spoofed ARP packets before they can permit more sophisticated attacks. Additionally, we demonstrated how our solution can be implemented on top of an existing network without interrupting current services.

Since traditional methods for security cannot easily transition to SDN, security features like NFG are needed to make SDN a viable alternative for network operators. SDN is quickly gaining ground in areas of detection and mitigation, which include countering port scans [13] and denial of service [14] attacks and detecting anomalies [15, 13]. All of these functions could eventually be included in a security system (like NFG) as modular components and allow network operators to pick and choose which security features our best suited for their networks without being relegated to a command line interface for each implementation. Such frameworks also have great potential to reduce the number of security middleboxes on networks.

While SDN may not remove all security middlebox requirements on government, campus, and industry networks, we see great potential to significantly reduce the number of such devices. Furthermore, Network Flow Guard (NFG) offers an excellent opportunity to

enhance network security with vendor-neutral solutions. Building on this work, we have released our NFGA security module, testbed, and ArpPoison software for further development, augmentation, and expansion via GitHub.

3.10 Segue

In this chapter and the last, the security solutions both use Pyretic [16], which is an excellent programming framework. However, this language is also limited. Since Pyretic sits atop the POX [31] controller, which is only enabled for OpenFlow 1.0 [42], programmers are only granted access to 12 packet header fields. Further complicating matters, Pyretic is now listed as deprecated. So, no further expansion of this programming language is expected.

Another drawback to the work in this chapter and the previous one is that network operators must manually terminate the security policy enforcements once they are activated. Resultantly, a traditional and tedious requirement is still not alleviated for network operators. Moreover, this requirement still presents opportunities for human error to create additional network errors as they delete clients from or add clients to their access control lists.

Hence, then next two chapters, address a new programming framework and a security policy transition framework respectively. The new programming framework and SDN controller allows programmers to maintain a greater level of abstraction for their applications while also achieving more robust management and security applications. Then, the security policy transition framework allows for automated revocation of policy enforcement so that network operators can focus on more complex tasks.

CHAPTER 4

RYURETIC: A MODULAR PROGRAMMING FRAMEWORK FOR RYU

4.1 Introduction

Software-defined networking (SDN) [10] allows for a single controller to orchestrate an entire network of switches, and OpenFlow [12] provides a single, vendor-agnostic interface for these devices. However, while OpenFlow [12] represents the *de facto* standard for communication between control and data planes, it lacks the abstractions needed for operators to focus more on their desired applications than on the network's inner-workings. As a result, controllers such as POX [31] and Ryu [17], to name a couple, were developed to communicate with OpenFlow switches via a southbound interface. Yet, even with these abstractions network operators are not completely shielded from the complexities of that are inherent in network application development.

The natural approach for these controllers then is to develop new programming interfaces. For POX, the Pyretic [16] framework was developed to create a simple, yet powerful, abstraction using packet matching fields from various network protocols as shown in Table 4.1. Using this framework, users can build multiple modules and then chain them together, either sequentially or in parallel, in a simple and intuitive manner. These features enabled our earlier research [73, 74], which we discussed in Chap. 2 and Chap. 3. For both of these solutions, we created a modular platform, called Network Flow Guard Coupler, using Pyretic in order to augment a simple, MAC-learning protocol running on an OpenFlow 1.0 [12] switch with additional security applications.

Table 4.1: Pyretic Fields

Ingress Port	Ether Src	Ether Dst	Ether type	VLAN ID	VLAN Priority	IP src	IP dst	IP Proto	IP ToS Bits	TCP/UDP src port	TCP/UDP dst port

Table 4.2: Ryu Fields

Ethernet			ARP									ICMP			
src	ethertype	dst	src_mac	dst_mac	src_ip	dst_ip	hlen	hwtype	opcode	plen	proto	code	csum	data	type
IPv4												UDP			
src	hdr_len	csum	flags	tot_len	option	offset	id	proto	tos	csum	ttl	src_port	dst_port	tot_len	csum
TCP										IPv6		DHCP			
src_port	dst_port	seq	ack	win_size	bits	urgent	csum	offset	option	Intentionally not shown					

Unfortunately, work for both Pyretic and POX has stalled over the past two years preceding this work. In fact, as of this writing, POX has yet to evolve beyond the incorporation of OpenFlow 1.0 functionality, and it is still limited to the 12 packet matching fields shown earlier in Table 4.1. These limitations proved a significant hindrance to our subsequent work. As a result, we turned to Ryu [17].

Ryu is a component-based programming framework for SDN development, and as shown in Table 4.2, Ryu gives programmers access to many more fields than does Pyretic [16]. Additionally, Ryu supports all current versions of OpenFlow [12], which includes version 1.5 as of this work, and it offers extensive documentation. Still, Ryu operates at a lower level of abstraction than Pyretic and lacks its modularity. This too proved challenging for our ongoing research and serves as the primary motivation for this work. Thus the main goal of this chapter is to propose Ryuretic as a modular, programming framework for the Ryu SDN controller. In doing so, we describe how Ryuretic is used to quickly couple multiple modules with existing applications to forward, redirect, mirror, drop, modify, or craft packets. Additionally, aspects of our Network Flow Guard Coupler from the previous two chapters are incorporated into the coupler we create for Ryuretic. Together, these features make Ryuretic an enabler for network operators seeking to incorporate traffic engineering, security, and other features into their network devices. Moreover, operators can implement proactive and/or reactive applications as we will soon discuss.

This chapter is outlined as follows. In Section 4.2, we introduce the Ryuretic framework and present a high-level view of its operation. Next, in Section 4.3, we discuss the components of the framework and how they enable modular abstractions. Then, in Section 4.4, we provide some motivating examples for modular application development using

Ryuretic. Further discussion and future work are presented in Section 4.5, and we conclude in Section 4.7.

4.2 The Ryuretic Programming Framework

To achieve its modular framework, Ryuretic provides an additional abstraction layer atop the Ryu controller. As shown in Fig. 4.1, this layer (the Ryuretic *coupler*) interprets instructions from multiple modules and passes them to the Ryu controller via its northbound interface. These modules can be either preexisting (e.g. the switch module) or created by the network operator and incorporated into the framework via Ryuretic's coupler. The Ryu controller then interprets these instructions and forwards them via its southbound interface to update flow [1] tables on its OpenFlow [12] switches.

The modularity of Ryuretic also means that programmers can easily create target-specific programs (e.g., load balancing, security, traffic engineering, etc.) separately, and then integrate these features into their switches via the Ryuretic coupler[2]. Researchers can also customize their security features to operate at specific layers of the OSI model and produce more specialized applications. Meanwhile, network operators benefit from an ability to choose and implement the network modules most applicable for their network's requirements.

[1] A flow consists of an ordered set of L2-L4 header fields

[2] Much the same as the Network Flow Guard Coupler does in Chap. 2 and 3

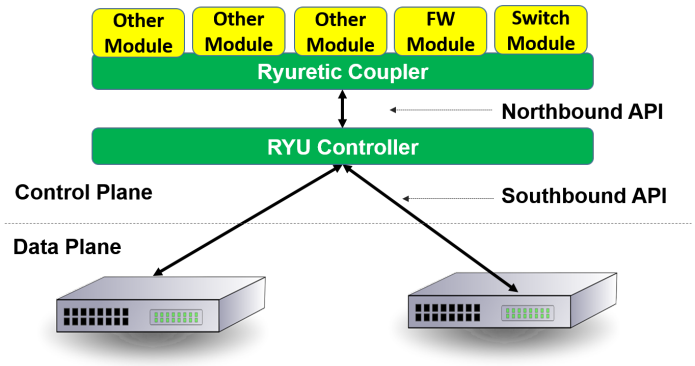


Figure 4.1: Modules Joined Via the Ryuretic Coupler

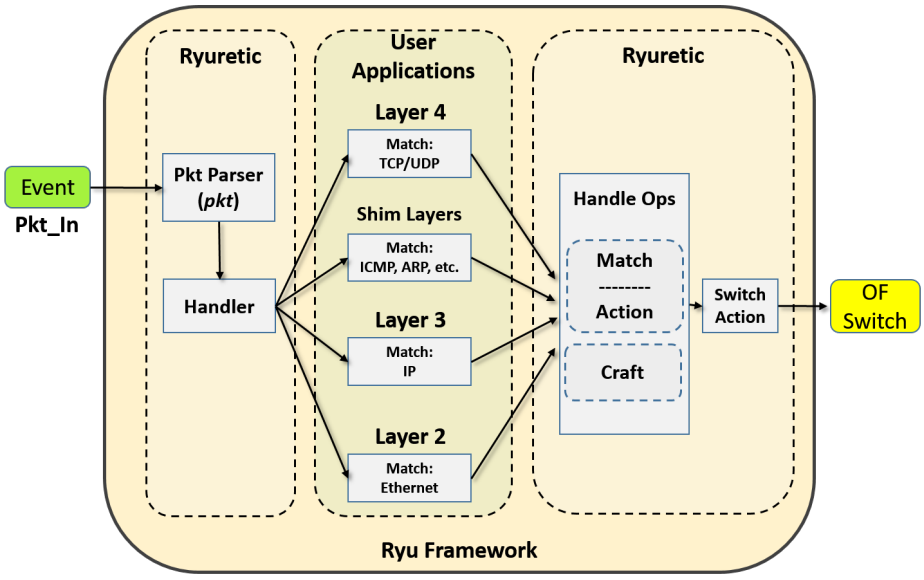


Figure 4.2: Ryuretic Framework

As mentioned previously, Ryuretic offers users the ability to proactively and reactively interact with packets. With proactive measures, Ryuretic does not wait for a packet to arrive. Instead, flow rules consisting of user provided match and operation specifications are immediately passed to switches at startup. Currently, Ryuretic proactively supports forwarding, dropping and redirecting matched packets.

To better articulate how Ryuretic reactively handles incoming packets, we utilize the abstract forwarding model shown in Fig. 4.2. When a packet arrives (i.e., a packet event occurs), Ryuretic first parses the packet and creates a packet object (*pkt*). This new object, as shown in Fig. 4.3, contains a timestamp, packet inport, datapath (e.g., switch ID), and packet header fields. Once *pkt* is built, it is passed to its corresponding handler as determined by its place in the OSI model (i.e., L2, L3, L4, or Shim Layer). This allows

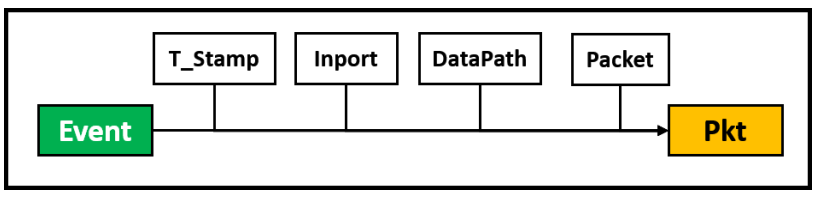


Figure 4.3: Ryuretic Packet Builder

Table 4.3: Ryuretic Fields: fields['*'].

Key	Summary							
keys	Key for fields object specifying packet header match fields**							
ptype	Packet type for crafted packets (Ether, ARP, IPv4, ICMP, etc.)							
**	**inport**	**ethtype**	**srcmac**	**dstmac**	**srcip**	**dstip**	**srcport**	**dstport**

Table 4.4: Ryuretic Operations: ops['*'].

Key	Summary	
idle_t	Key for fields object specifying packet header match fields**	
hard_t	Packet type for crafted packets (Ether, ARP, IPv4, ICMP, etc.)	
priority	Set priority for match-action rules	
op	fwd	Default forward packet setting
	drop	Drop matched packet flows
	mir	Mirror matched packet flows to specified port
	redir	Redirect matched packet flows to another port
	mod	Modify packet headers and redirect flows to another port
	craft	Create and send new packet
newport	Destination port for mir, redir, mod, and craft operations	

programmers to better target specific protocols in their applications. Consequently, each *pkt* object also contains the header information from the lower layers (i.e., if a TCP *pkt* is built, then it will also contain metadata for IP and Ethernet). Within each handler, the network operator calls network applications to return the hashes (*fields* and *ops*) and then passes them to *match* and *actions* objects required for the Ryu platform. So, when a module is created, the user can choose to return specific match fields (*fields*) and their operation parameters (*ops*). Both contain keys that map to specific fields in each object. The keys for *fields* are shown in Table 4.3 while the keys for *ops* are summarized in Table 4.4. When a user creates a new module, only the keys that hash to specific matching packet header fields need be set in the *fields* object. Similarly, operation requirements (e.g., idle timeout, hard timeout, priority, action, and new port) are specified in *ops*.

When *fields* and *ops* are returned to the coupler, it creates hashes (*match* and *action*) that are passed by Ryu to its OpenFlow switches. However, the user's only requirement in this process is to instantiate their desired network modules to evaluate *pkt* and then

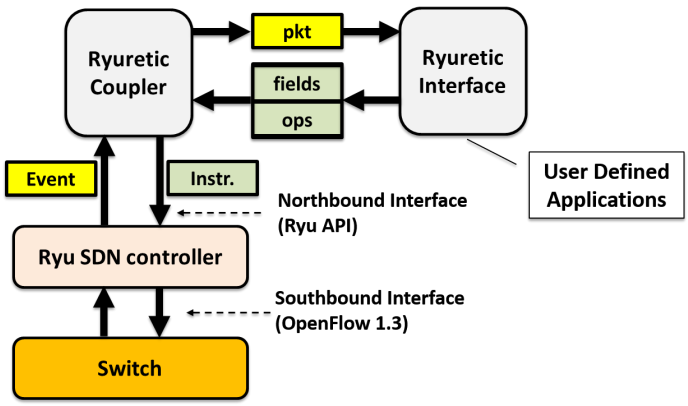


Figure 4.4: Ryuretic Controller

update *fields* and *ops* to implement their desired flow rule. This process greatly simplifies modular application development for network operators since they are no longer required to implement Ryu specific programming constructs. Instead, new application modules can focus purely on packet content (provided by the *pkt* object).

Fig. 4.4 essentially summarizes the actions that occur within the Ryuretic framework. Anytime a match-action rule is not found in the OpenFlow switch, the packet header is forwarded to the Ryu controller. The controller then generates an event, which the Ryuretic coupler receives. The coupler then parses the provided information from the controller and generates a packet, and that packet is passed to the Ryuretic interface for processing. Based on the network operator's defined applications, the Ryuretic interface returns two objects (fields and ops) back to the Ryuretic coupler, which translates these objects into instructions for the Ryu controller. The Ryu SDN controller then passes its instructions to the switch via its southbound interface.

4.3 Ryuretic Components

Within the Ryuretic framework, examples of metadata collected from arriving packets include timestamp, ingress port, datapath, and packet headers. At the highest level, programmers are able to develop SDN solutions as class methods and call them via the Ryuretic coupler module. When an event occurs, Ryuretic builds a *pkt* object using its packet parser

library (*Pkt_Parse*) and then passes it to applicable modules. This was depicted earlier in Fig. 4.2. Likewise, Fig. 4.3 shows how the packet parser assembles the *pkt* object that is subsequently passed to selected network applications, which are either created or obtained by network operators. We now provide an in-depth overview of the components that make the Ryuretic framework possible.

4.3.1 Coupler

The coupler component consists of two modules (i.e., the Ryuretic interface and the Ryuretic backend or coupler). Together, these modules serve as the heart of the Ryuretic framework. Additionally, the Ryuretic backend imports both standard Ryu libraries and additional Ryuretic dependencies. These currently include *Pkt_Parse* and *switch_mod*, which are discussed in the following subsections. However, within the Ryuretic interface (*Ryuretic_Intf.py*), network operators can input code snippets at specified locations to support their design goals. These locations are indicated as #[1] through #[5] within the interface file. In the first location, the user imports desired libraries (if needed). The default import is shown in location #[1] of Listing 4.1; however, the user can create and call other libraries from this location as well.

Listing 4.1: Coupler Import

```
1 #[1] User can add def or create their own file from
2 ryu.app.NFG import NFG
```

Listing 4.2: Coupler Object Creation

```
1 #[2] Add Additional modules or object variables
2 self.NFG = NFG()
```

Listing 4.3: Proactive Flow Rule Creation

```
1 #[3]Insert proactive rules using format below.
2 #Options are drop or redirect, fwd is default
3 def get_proactive_rules(self,dp,parser,ofproto):
4   # return None, None
5   fields, ops = self.honeypot()
6   return fields, ops
```

The second location (shown as #[2] in Listing 4.2) allows the user to instantiate their application as an object, if using another file, or to create additional variables. However, this is already set by default within the Ryuretic interface module. In the third location (see #[3] in Listing 4.3), the user can provide proactive instructions to be installed shortly after the Ryuretic backend (*coupler*) module performs its initial switch configurations to have packets forwarded to the controller. As a result, these instructions are applied to the OpenFlow switch at startup. The default value for this section is already defined to return *None*, *None*; however, as shown in the snippet, these can be commented out and replaced with method calls to load the hashes, *fields* and *ops*. These rules are then permanently installed to the OpenFlow switch, which alleviates the controller's workload.

Otherwise, when a packet-in event occurs, Ryuretic reactively responds by having the coupler's *initial_event* method call the packet parser to create a *pkt* object. The coupler then determines which OSI layer (i.e., L2, L3, L4, or shim layer) applies to *pkt* and forwards it to the appropriate handler as shown in Fig. 4.2. Here is the fourth location where users add code within the Ryuretic interface. For instance, if the user plans to call on a security module and apply it to TCP packets, then the user may include code as shown in location #[4], depicted in Listing 4.4. Here we also note that only the code located on lines 3 and 7 need be modified. In this example, line 3 holds the standard default method call for setting *fields* and *ops*, which is required if no other module is called to set the *fields* and *ops* objects. On line 7, a simple firewall method from the next location we will discuss is called to filter IP packets. Note that the default method is removed or commented out.

Listing 4.4: Coupler Reactive Object Creation

```
1 #[4] Use below handles to direct packets to modules...
2 def handle_eth(self,pkt):
3   fields, ops = self.default_Field_Ops(pkt)
4   self.switch_mod(pkt,fields,ops)
5    ...
6 def handle_tcp(self,fields,ops):
7   fields,ops = self.Simple_FW(pkt))
8   self.switch_mod(pkt,fields,ops)
```

Listing 4.5: Multiple Modules

```
1 def handle_ip(self,pkt):
2   fields0, ops0 = self.Simple_FW(pkt)
3   fields1, ops1 = self.TTL_Check(pkt)
4   #Determine highest priority fields,ops pair
5   xfields = [fields0, fields1]
6   xops = [ops0, ops1]
7   fields, ops = self._sel_FldOps(xfields,xops)
8   self.switch_mod(pkt,fields,ops)
```

Ryuretic can also accommodate multiple modules acting on the same *pkt*. The user needs only modify the names of *fields* and *ops* for each application call and store them in the *xfields* and *xops* lists, as shown in Listing 4.5. These lists are then passed as arguments to the coupler's selection method as depicted on line 7, which selects and returns the appropriate *fields* and *ops* hashes based on the network operator's priority. This requires that the network operator assign the highest priorities to the most restrictive rules. Subsequent operation handlers use *fields* and *ops* to build *match* and *actions* objects (used by Ryu) before passing them to the switch module where flow instructions are created and ultimately pushed down to OpenFlow switches via the Ryu controller.

In the fifth and final location where users modify the Ryuretic interface file, users can paste or create their own network applications. These applications can be proactive or reactive. Proactive methods, as seen in Listing 4.6, do not receive a *pkt* object as an argument, however, they do return objects, *fields* and *ops*, to define the proactive flow rules for switch initialization. Note also that objects *fields* and *ops* are initially declared as empty objects.

Listing 4.6: Proactive Flow Rule Definition

```
1 #[5] Add user created methods below
2 def honeypot(self):
3   # Redirect all IP traffic from srcip to port 2
4   fields, ops = {}, {}
5   fields['keys'] = ['ethtype', 'srcip']
6   fields['ethtype'] = 0x0800
7   fields['srcip'] 192.168.0.4'
8   ops['priority'] = = '100
9   ops['op'] = 'redir'
10  ops['newport'] = 2
11  return fields, ops
```

In contrast, reactive methods require a *pkt* object as an argument and may or may not return *fields* and *ops* hashes depending on user requirements as specified in location #[4]. The returned values instruct Ryuretic on how long the new flow rule is to be maintained in the switch's flow table and assigns the rule a priority. Furthermore, ops['op'] instructs the Ryuretic coupler to either forward, drop, mirror, or redirect the packet or even craft a new one. A very simple method creation example is shown in Listing 4.7. In this case, the programmer writes a definition to receive the *pkt* hash and then returns the objects, *fields* and *ops*, to the calling method. Consequently, if a method does not return *fields* and *ops*, then Ryuretic requires that these hashes be obtained via the *default_Field_Ops(pkt)* call or other application within the appropriate handler as previously shown in Listing 4.4. Additionally, users may still develop their modules separately in new libraries and import them as demonstrated in Listings 4.1 and 4.2. However, these libraries must be created as subclasses of the Ryuretic *coupler*.

Listing 4.7: Ryuretic Method Creation

```
1  #[5] Add user created methods below
2  def Simple_FW(self,pkt):
3    fields, ops = self.default_Field_Ops(pkt)
4    #blocking w3cschools and facebook
5    if pkt['dstip'] in ['141.8.225.80', '173.252.120.68']:
6      print "W3Cschools or Facebook is not allowed"
7      #tell controller to drop pkts destined for dstip
8      fields['keys'] = ['dstip']
9      fields['dstip'] = pkt['dstip']
10     ops['priority'] = 100
11     ops['op'] = 'drop'
12   return fields, ops
```

4.3.2 Packet Parser

In addition to the fields shown in Table 4.2, the *Pkt_parse* module also adds a timestamp, switch input port number, and datapath information to a created *pkt* object. For Ryuretic, the timestamp always indicates the first instance of a *pkt* object in the Ryuretic framework. That is to say that *pkts* are stamped at the time the controller receives them from

Table 4.5: Ryuretic Pkt Object: pkt[‘*’]

Ethernet			ARP									ICMP			
src	ethertype	dst	src_mac	dst_mac	src_ip	dst_ip	hlen	hwtype	opcode	plen	proto	code	csum	data	type
IPv4												UDP			
src	hdr_len	csum	flags	tot_len	option	offset	id	proto	tos	csum	ttl	src_port	dst_port	tot_len	csum
TCP										IPv6		DHCP			
src_port	dst_port	seq	ack	win_size	bits	urgent	csum	offset	option	Intentionally not shown					

the switch. With the timestamp, users can calculate round trip time (RTT) estimates, inter-packet arrival times (IAT), etc. Like Pyretic, Ryuretic allows users to extract header fields via the *pkt* object's keys. For instance, pkt[‘srcip’] will return the IP address of the packet's source network device while pkt[‘dstmac’] will provide the MAC address of the packet's intended destination. Table 4.5 provides some of the fields available to the user. As of this work, header fields for IPv6 and DHCP packets are withheld for future work. By including switch identification information in *pkt*, Ryuretic also supports the orchestration of multiple switches. Likewise, the inclusion of the inport allows network operators to quickly build port-MAC-IP associations, which are helpful for building targeted security features.

Additionally, each *pkt* contains all needed information to allow network operators to work with packets transiting L2-L4 of the OSI model. The Ryuretic coupler uses this feature to segregate incoming packets into their appropriate OSI layer handler. For instance, if an IP packet arrives, the created *pkt* will also contain Ethernet fields as well as the IP fields. If a TCP packet arrives, then the corresponding object will contain TCP, IP, and Ethernet information. This feature allows network researchers to implement an entire gambit of features impacting multiple layers of the OSI model for each packet regardless of the layer for which it is intended.

4.3.3 Switch Module

The switch module (switch_mod) is a modified version of the mac-learning switch originally provided with the Ryu controller software. However, its event declarations and its direct interaction with OpenFlow switches have been removed. It now simply builds and maintains a simple, MAC-learning table which the coupler accesses before generating its

own flow table updates. As a result, this file requires no additional modifications, but it must be included with the Ryuretic framework in order to maintain basic switch functionality.

4.4 Ryuretic Programming Examples

To better demonstrate Ryuretic as an enabler for network or security applications, we examine two, simple use cases for this framework. In the first, we implement a stateful firewall. Then, in the second, we implement an unauthorized NAT detection solution.

4.4.1 Simple Stateful Firewall Application

One feature network operators may be interested in is isolating their network from outside entities. This can be accomplished by using SDN to implement a stateful firewall. Doing so can ensure that outside network devices are unable to establish an IP connection with local hosts unless a host initiates the connection. Hence, two scenarios occur in our model and shown in Fig. 4.5. In the first scenario, a host generates a *get* request for an external server, and the external IP is stored. When the external device responds its stored IP is found, and its packets are allowed. In the second scenario, an outside IP attempts to contact a host on the subnet without first being contacted. In this scenario, the packet is dropped.

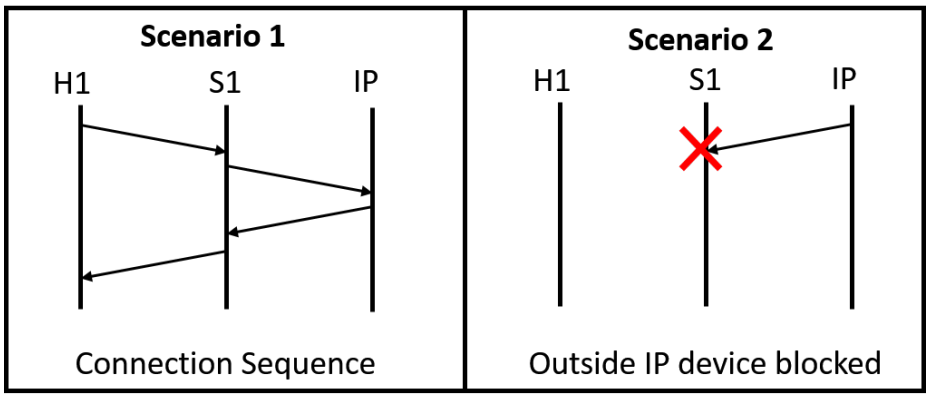


Figure 4.5: Stateful Firewall Event Sequence

Listing 4.8: Ryuretic Object Creation

```
1 #[2] Declare any global variable here
2 self.stat_Fw_tbl = {}
```

To implement this application, we first create a table in location #[2] (see Listing 4.8). In this example, there is no requirement to add any imports. We then create the method that will handle Ryuretic's *pkt* object in the location #[5] (see Listing 4.9). As with other methods, line 2 immediately sets *faults* and *ops* to their default values. As indicated on line three of Listing 4.9, the port connecting to external network traffic is port 10. Therefore, all other ports connected to the switch are considered local.

Listing 4.9: Ryuretic Method Creation

```
1 # [5] Add user created methods below
2 def Stateful_FW(self,pkt):
3   fields, ops = self.default_Field_Ops(pkt)
4   if pkt['input'] != 10:
5     if self.stat_Fw_tbl.has_key(pkt['srcip']):
6       if len(self.stat_Fw_tbl[pkt['srcip']]['dstip']) > 4:
7         self.stat_Fw_tbl[pkt['srcip']]['dstip'].pop(3)
8         self.stat_Fw_tbl[pkt['srcip']]['dstip'].append( \
9                   pkt['dstip'])
10    else:
11      self.stat_Fw_tbl[pkt['srcip']]={'dstip':[pkt['dstip']]}
12      return fields, ops
13  else:
14    if self.stat_Fw_tbl.has_key(pkt['dstip']):
15      if pkt['srcip'] in stat_Fw_tbl[pkt['dstip']]['dstip']:
16                  return fields, ops
17    else:
18      fields['keys'] = ['srcip','dstip']
19      fields['srcip'] = pkt['srcip']
20      fields['dstip'] = pkt['dstip']
21      ops['priority'], ops['op'] = 100, 'drop'
22      ops['hard_t'], ops['idle_t'] = 20, 4
23  return fields, ops
```

If an IP packet originates from any of these ports, then the application first checks to determine if the source IP (*srcip*) has already been entered into the state firewall table (*stat_Fw_tbl*). If not, the initial *srcip* and destination IP (*dstip*) values are entered. Otherwise, the table is updated by appending a new (*dstip*) to the list contained in *stat_Fw_tbl*. To prevent the table from growing too large, the table also limits the number of *dstips* that

may be stored at one time by dynamically dropping IPs using the 'first in first out' (FIFO) queuing method.

When a packet-in does occur on port 10, the application then performs a reverse table lookup to determine if the *dstip* (a local host address) has the packet's *srcip* stored in the state firewall table. If so, the packet traverses the switch normally. If not, the application modifies *fields* and *ops* to drop the packet. In this case, *fields* is set to match on the packet's *srcip* and *dstip*. Then as shown in lines 19-21, *priority*, *op*, *hard_t*, and *idle_t* keys are set. In this case, the corresponding flow update has a priority of 100, a rule to drop matched packets, and an arbitrary duration of no more than 20 seconds total or four seconds if the IP address becomes idle.

Listing 4.10: Ryuretic FW Method Call

```
1 def handle_tcp(self,pkt):
2   #fields, ops = self.default_Field_Ops(pkt)
3   fields,ops = self.Stateful_FW(pkt)
4   self.switch_mod(pkt,fields,ops)
```

Once updates to Ryuretic's method library are complete, the user then updates the coupler module as follows. First, if the method library is used, the code snippets for sections #[1] and #[3] of the Ryuretic interface module remain the same as shown in Listings 4.1 and 4.3. Location #[2] must have the global variable *stat_Fw_tbl* instantiated as was done in Listing 4.8. The only other change to the interface file is in location #[4] (see Listing 4.10). Note that the default method for setting *fields* and *ops* is commented out. Instead, it is replaced with a call to the previously created Stateful_FW module. Note also that this application is called from the TCP handler within the coupler, so only TCP packets are affected by this security feature.

4.4.2 Unauthorized NAT device

Unauthorized network address translation (NAT) devices can also compromise local networks. One way to detect these devices is to monitor IP packets for decremented time-

to-live (TTL) header fields [75]. In this example, the *nat_detect* module in Listing 4.11 inspects the TTL field of each IP packet passing through the switch, which is something we could not accomplish with Pyretic due to its limited number of header fields. Now, building the NAT detector module is a fairly straightforward process. We first observe that most network devices have TTL values of 64 or 128. If hosts are directly connected to the switch, then the switch should detect one of the prior values. However, if these devices are connected to a NAT device, then the switch will detect the decremented TTL.

Listing 4.11: Ryuretic NAT Detection Method Creation

```
1  #Block IP packets with decremented TTL
2  def TTL_Check(self, pkt):
3    fields, ops = self.default_Field_Ops(pkt)
4    if pkt['ttl'] == 63 or pkt['ttl'] == 127:
5      print "XxXxXx  NAT Detected  xXxXxX"
6      #drop all packets from port with TTL decrement
7      fields['keys'] = ['inport']
8      fields['inport'] = pkt['inport']
9      ops['priority'] = 100
10     ops['op'] ='drop'
11   return fields, ops
```

As before, we first load *fields* and *ops* using the default *_Field_Ops* method. We next evaluate the TTL field of the provided *pkt*. If the TTL is equal to 63 or 127, then a TTL decrement has occurred and an unauthorized NAT device is detected. As a result, the application modifies *keys* in the *fields* hash to indicate that the *inport* is the only match requirement. It also updates the *inport* key with the port number of the connected NAT device. It then updates the *ops* hash to indicate that packets from the offending port should be dropped with a priority of 100. In this example, we do not provide a duration for the flow rule, so the rule is not installed on the OpenFlow switch. The result is that all packets continue to be sent to the controller for decision. Ideally, the network operator would install the flows to reduce the burden to the controller. Of course, if the packet passes the TTL test, then the default *fields* and *ops* objects are returned to the coupler for basic routing. Now that the method has been created in the method library, the user needs only update the TCP handle in location #[4] of the coupler module as demonstrated previously.

4.5 Discussion

While Ryuretic was initially created to support our own work, we quickly realized that network operators, researchers, educators, and students could all benefit from an intuitively simple tool for network application development. Moreover, Ryuretic's abstractions obfuscate the complexities and modularity constraints of Ryu [17] while allowing users to handle packet header fields in a manner similar to Pyretic [16]. Ryuretic is less generalized and slightly more restrictive than Pyretic (lacking the ability to combine rule sets via cross product); yet, it does not generate excess rules either. Furthermore, it grants users access to all current OpenFlow [12] protocols.

Consequently, a more recent development with Pyretic adds greater value to our work. While some researchers (ourselves included) awaited updates to the Pyretic platform to utilize OpenFlow 1.3 and higher, support for Pyretic fell off over the last two years. Regrettably, as of October 2016[3], the Pyretic programming framework is listed as deprecated and no longer supported.

With Ryuretic, operators need only develop a rudimentary understanding of the Python programming language to utilize its lists, dictionaries, and methods to implement their network applications. Likewise, the use of the *pkt*, *fields*, and *ops* objects in Ryuretic greatly simplifies application development; however, users must still understand network operations, header fields, and the OSI model to intuit and handle network packets appropriately. Ryuretic is already enabling the development of modular security applications to detect and mitigate rogue access points (RAPs) and unauthorized network address translation (NAT) devices in SDN environments.

Additionally, Ryuretic is already being used to teach software-defined networking in one graduate level networking course offered by the Georgia Institute of Technology, and Ryuretic is currently facilitating at least one master's project at the Universiti Putra Malaysia

[3]Date is an approximation based on results obtained from `https://web.archive.org/web/*/http://frenetic-lang.org/pyretic/`. May 2016 was the last month recorded that showed Pyretic as still being supported.

(UPM). Also, as shown in Fig. 4.6, Ryuretic Labs, which is available through GitHub[4] provides instructions for setting up Ryuretic along with instructional project-oriented lab exercises.

4.6 Future Work

Presently, we are working to expand Ryuretic's packet parser to include IPv6, DHCP, and LLDP fields in its *pkt* object. There is also room to parallelize functions within Ryuretic to improve performance when handling multiple modules. With the packet crafting feature allowed by Ryuretic, we are also seeking to enable new ways for Ryuretic to communicate with trusted entities on the network. We believe the incorporation of such communication protocols will enable more robust network management and security options for network operators. Still, as it is, we believe that the Ryuretic framework serves as a useful tool for users seeking to develop, modular, network applications. The Ryuretic platform and its subsequent versions and applications are also available to users via GitHub[5]

[4] Available at https://github.com/Ryuretic/RyureticLabs.
[5] https://github.com/Ryuretic/Ryuretic

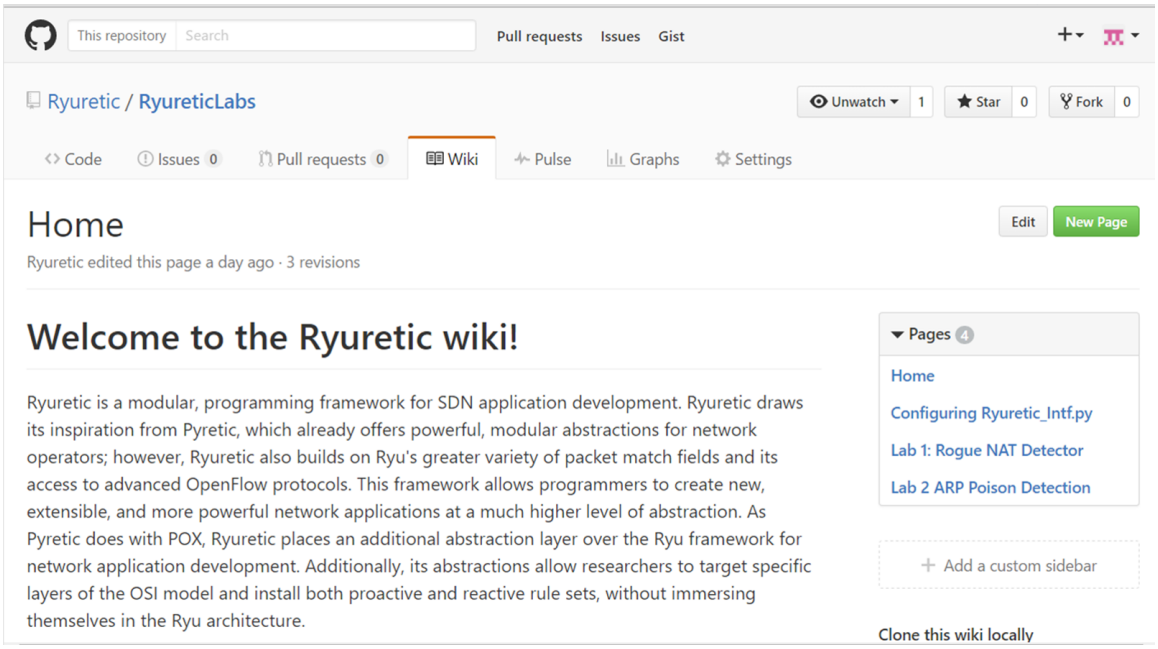


Figure 4.6: Ryuretic Labs

4.7 Conclusion

With OpenFlow providing a vendor agnostic platform for SDNs and enabling the orchestration of numerous switches, programmers are able to implement novel network applications for security and traffic engineering. Yet, network operators still need abstractions to enable network application development without its associated complexities. Towards this effort, we offer Ryuretic as a novel, modular, SDN-based framework that enhances the Ryu controller platform with abstractions that greatly simplify application development. In addition, Ryuretic allows network operators to selectively interact with packets from layers L2, L3, and L4 of the OSI model using both proactive and reactive measures. Additionally, Ryuretic limits operator involvement to evaluating packet headers, choosing match fields, and selecting a desired operation. Furthermore, Ryuretic's modular nature offers network operators an extensible platform for providing further network enhancements.

4.8 Segue

One of the limitations that Ryuretic still fails to address for network operators is an ability to automate the updating of policy enforcements. As a result, network operators are still creating policies that require their manual intervention to change. However, Ryuretic's packet crafting feature serves as a key enabler to create such an option. In the following chapter, we will incorporate a virtual entity called a Trusted Agent to serve as an intermediary between the network operator and the SDN controller to better automate network security policy transitions and avoid network operator errors.

CHAPTER 5
SECURITY POLICY TRANSITION FRAMEWORK FOR SOFTWARE-DEFINED NETWORKS

5.1 Introduction

Software-defined Networking (SDN) [10] allows for a single controller to orchestrate an entire network of switches, and OpenFlow [12] provides a single, vendor-agnostic interface for these devices. Additionally, other frameworks, like Pyretic [16] and Ryuretic [5], provide abstractions that shield network operators from the complexities inherent in network application development. However, operators also require frameworks to reverse security measures (e.g. blocking ports and redirecting or dropping traffic flows) once they are triggered. Unfortunately, once a system is flagged for a security (or compliance) violation, revoking the implemented security measure in these solutions is not possible without resetting the controller or requiring the network operator to manually reinstate the client's privileges via a script or external command.

As Kim et al. [1, 2] observe, network operators may already be responsible for as many as 18,000 network configuration changes per month (much of which deals with adding, modifying, or deleting entries in access control lists) on traditional networks. Additionally, with each configuration comes the opportunity to introduce a new network error. Furthermore, managing and maintaining ACLs that can contain nearly 10,000 entries and see as many as 4,000 changes per year[1] is a burdensome challenge [1]. Moreover, if network operators must manually revoke triggered security measures on SDN controllers, then this too becomes another cumbersome task for many network operators who lack actual programming experience as Kim et al. [2] observe.

[1] Discussed values pertain to a study involving two universities.

Hence, security policy transition frameworks, such as the one we have implemented and describe in this chapter, can greatly assist network operators and their clients by automating security policy transitions. Such a framework offers substantial benefits. First, it reduces the network operator's configuration requirements that subject the network to additional errors and delay the execution of more critical tasks. Second, clients receive automatic notification of their violation and instructions for regaining their network privileges. Third, it eliminates erroneous trouble tickets by informing both clients and administrators of the violation. Finally, depending on the violation and validation requirement, it reduces the total time required to reinstate a client's network privileges. Having triggered a security policy, the client need only enter a *passkey* into a provided web interface to regain their privileges. Of course, while implemented using OpenFlow, this framework could be built atop other architectures as well. It is also easily implementable with virtual switches.

This chapter is outlined as follows. We first discuss the motivation for our framework in Section 5.2 and related work in Section 5.3. Next, in Section 5.4, we discuss our framework. Then, we discuss our test environment in Section 5.5 and an example use case in Section 5.6. We then offer further discussion in Section 5.7 and conclude in Section 5.8.

5.2 Framework Motivation

The primary motivation of this work is to reduce network operator involvement with repeated network reconfigurations. We now present a few examples for how automating security policy transitions alleviate this burden. Also, in cases where the process cannot be completely automated, we suggest that a less-skilled help desk attendant be utilized to further reduce network operator involvement. For instance, patch compliance and policy violations can potentially be completely automated, while infected computers that require operating system reinstalls can potentially be handled by help desk personnel. A high-level view for this framework is depicted in Fig. 5.1.

In our framework, the network operator sets the security policies for the controller as

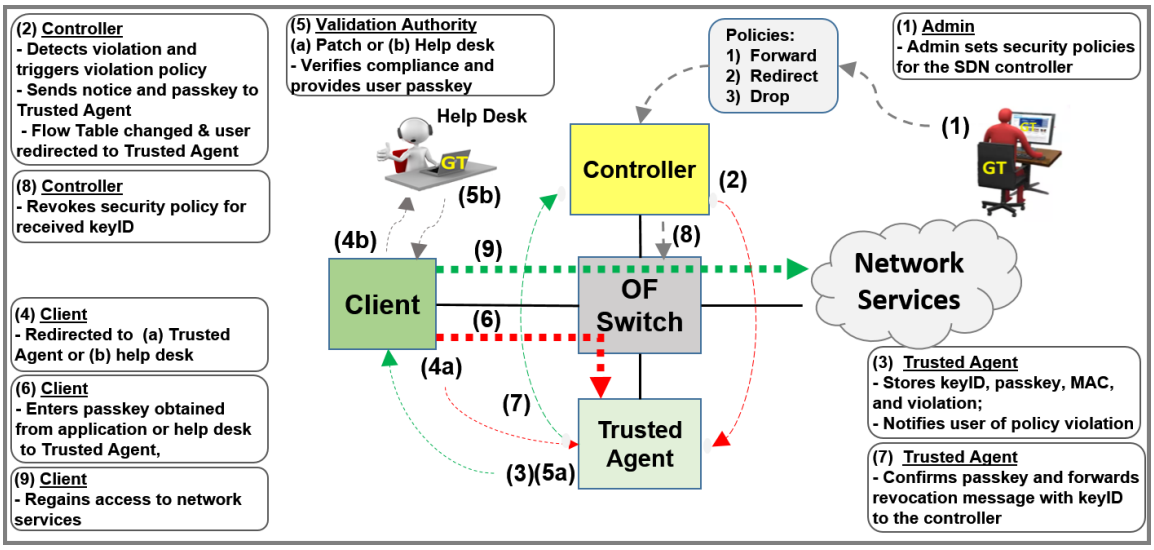


Figure 5.1: Security Policy Transition Framework

shown in (1) of Fig. 5.1. Then, as shown in (2), when the controller detects a violation and triggers a security measure, the controller informs the Trusted Agent and updates flow tables within the OpenFlow switch to redirect the client's future traffic to the Trusted Agent. In (3), the Trusted Agent accepts and stores the client's *keyID*, *passkey*, MAC address, and violation. Thus, for each of the cases mentioned above, the client is flagged and redirected to a "walled garden" where our Trusted Agent presents the client with a web interface. This interface provides the client notice of their violation along with instructions for gaining compliance. For instance, if the client is flagged for patch compliance, then the Trusted Agent can make the patch available for download (4a) and (5a). Then, once the software is installed and validated, the client receives a *passkey* from a validation authority (5a or 5b) which can subsequently be entered into the web portal. Upon entering a correct *passkey* (6), the Trusted Agent communicates the revocation request to the controller (7), and the client's network privileges are reinstated (8).

Similarly, in cases where clients inadvertently (or overtly) violate network security policies (e.g., DoS, scanning, ARP poisoning, etc.) and get flagged by SDN security protocols, then the Trusted Agent can provide client policy training to the client. The client would then be required to complete and pass the course, digitally sign an Acceptable Use Policy

(AUP), and cease such actions in the future. The *passkey* can also be provided with the certificate of completion. Additionally, a network operator can choose what level of involvement they wish to have in this process. For instance, they may want to take actions based on a first occurrence, a third, etc. In a corporation or government office, the network operator might require the first line supervisor to login and acknowledge the incident before granting the certificate.

Another common case occurs when clients are flagged for a virus requiring their system to be re-imaged. For such cases, a validation authority, such as a help desk can easily provide this service or verify that specific actions were completed. A *passkey* can be provided once the action is confirmed. For all of these examples, the client regains network privileges without involving the network operator.

5.3 Related Work

The goal of security management is to prevent local networks from being sabotaged (intentionally or unintentionally) by controlling access to network resources in accordance with organizational guidelines. This control is often implemented by systems that monitor client logins and refuse access to those who fail to authenticate or lack authorization. Accordingly, various methods for controlling network access exist in traditional networks and SDNs. Traditional security management methods include access control lists (ACL), client IDs and passwords, and terminal access controller access control (TACACS) [76]. These are indeed effective tools for enforcing prearranged policies on system networks. However, these policies are often reconfigured by network operators each time a security violation occurs or when a client (who triggered the security measure) regains approval to be reinstated. Additionally, protocols like 802.1X [77] will shut down ports if they detect unassigned devices connected to them, but reactivating these ports is often left to the network operator to resolve via a trouble ticket. Hence, these solutions place considerable configuration burdens on network operators, add additional software and hardware costs,

and lack an automated security policy transition framework for reinstating clients.

SDN solutions have also developed in recent years to assist network operators with security management. For instance, SNAC [78] provides network operators with a web-based policy manager for network monitoring. It also includes a client interface and a flexible policy definition language for device configuration and event monitoring. Even so, SNAC does not provide an automated process for reinstating client privileges once lost. Another solution assists network operators with migrating firewall ACLs from traditional networks into an SDN [79]. The process evaluates policy rules automatically [79], [80]. Ethane [81], a precursor to SDN, provides a centralized network architecture with identity-based access control that allocates IP addresses as IP-MAC-port associations. In this environment, clients authenticate via a web-form, and their packets are then reactively evaluated by the controller for policy compliance. FlowNAC [82] drops web-based authentication in favor of a modified 802.1X framework supporting extensible authentication protocol over LAN (EAPoL-in-EAPoL) encapsulation. However, supplicant (client) software is necessary for FlowNAC.

Kinetic, formerly known as Resonance, is a domain specific language (DSL) that offers an OpenFlow-based dynamic access control system [2]. It uses network alerts to support continuous monitoring and per interface policy control to automate dynamic security policies. Additionally, Kinetic verifies that prescribed changes align with operator requirements by employing a finite state machine (FSM) having states that correspond to distinct forwarding behavior [2]. Transitions within the FSM are controlled by Kinetic's event handler, which monitors for events and triggers policy updates. However, Kinetic relies on the network operator to supply the events that trigger its policy changes[2]. Also, since Kinetic is built atop the Pyretic [16] programming language and POX [31], it is limited to OpenFlow 1.0 [12] and has access to only 12 packet header match *fields*. It also does not include an automated framework for transitioning between security measures.

[2]The Pyretic runtime can also be utilized.

These solutions all represent great strides towards better and more intuitive interfaces that simplify the application development process, yet they still do not provide a framework for automating the process for revoking security measures once implemented. Our work is unique in its focus on security policy transitions within SDN environments, which improves turnaround times for reinstating network clients while reducing network operator workloads. Like Kinetic, our solution implements an event listener (aka Event Handler); however, it works with a trusted entity (a.k.a. Trusted Agent) to determine when an activated security action should be revoked. Additionally, our controller assumes responsibility for implementing security measures, but then relies on its Trusted Agent to provide notification for when the measure can be revoked.

5.4 The Framework

The security policy transition framework introduced in this chapter uses Ryuretic [5] for its SDN controller applications. Ryuretic [5] is a domain specific language offering a modular framework for application development atop the Ryu [17] controller. It also provides an intuitively simple format for network operators to select header fields within a packet (pkt[*]) and then specify what operation (*ops[*]*) occurs when a match (*fields[*]*) is found. This platform also allows programmers to craft their own packets, which is utilized to establish a communication channel between the SDN (Ryuretic) controller and its Trusted Agent using ICMP packets. This communication channel is then used to submit policy enforcement updates or revocation requests. This is discussed in greater detail in Subsection 5.4.3. Additionally, this communication allows both the Ryuretic controller and the Trusted Agent to maintain corresponding state tables as we will also discuss in Subsection 5.4.3. These and the other components comprising the controller and Trusted Agent modules (shown in Fig. 5.2) are discussed in the following subsections.

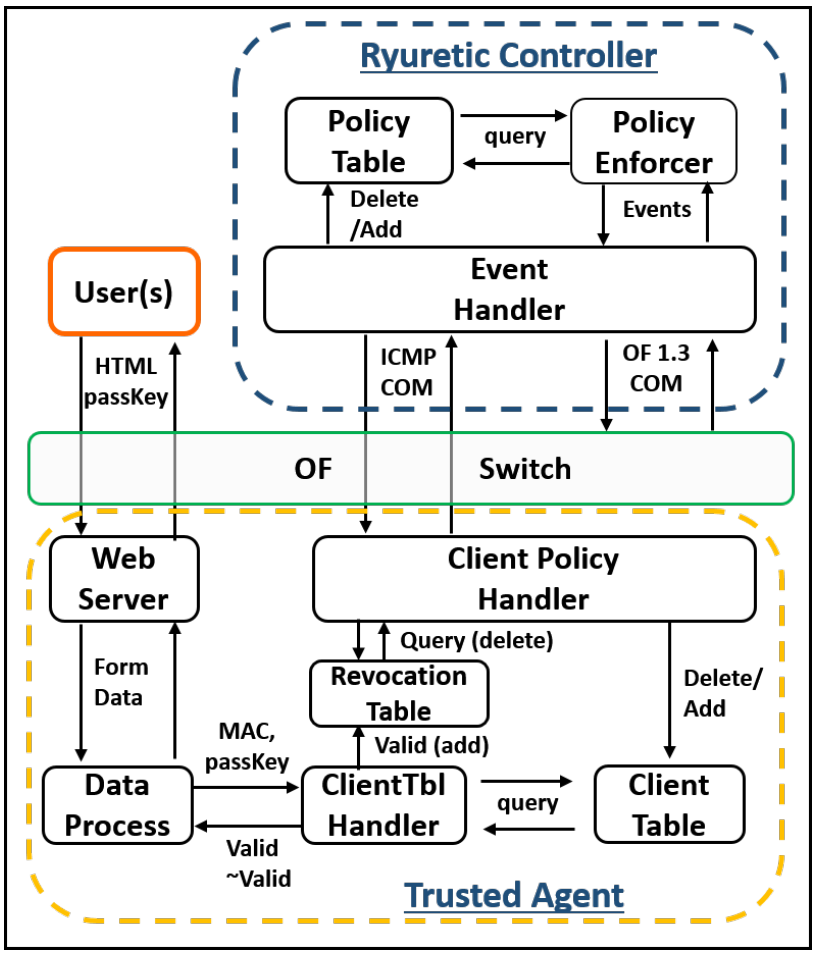


Figure 5.2: Security Policy Transition Framework Components [30]

5.4.1 Controller

As shown in Fig. 5.2, the Ryuretic controller for this framework is an SDN controller comprised of an Event Handler, a Policy Enforcer, and a Policy Table. These components are implemented in Ryuretic [5], which serves as an abstraction layer residing above the Ryu [17] controller and supporting OpenFlow 1.3 [12]. With Ryuretic, network operators can choose to forward, drop, mirror, redirect, modify, or craft packets based on match parameters that they define via objects.

As shown in Fig. 5.3, when a packet-in event occurs in the Ryu [17] controller, the Ryuretic Coupler generates a packet object (*pkt*) that is forwarded to the Ryuretic Interface. This is where the network operator specifies their policies. Based on these policies, the

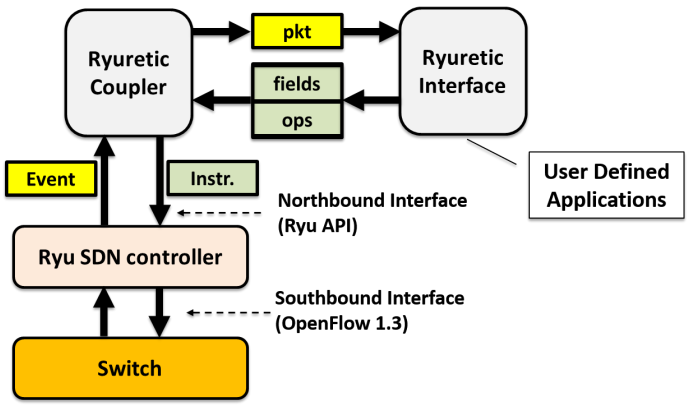


Figure 5.3: Ryuretic Controller

network operator modifies two objects (i.e., *fields* and *ops*) to specify the match-action rules that are to be performed on the switch. These objects are then interpreted by the Ryuretic Coupler and forwarded as instructions to the Ryu controller, which installs the rules to the switch.

Event Handler

The Event Handler serves as the primary interface for the controller, responding to network events from the switch, security events from the Policy Enforcer, and security policy transitions from the Trusted Agent. It also handles insert and delete messages for the controller's Policy Table to maintain state for each connected client. Once a client is flagged, the Event Handler creates a unique *passkey* and a *keyID* for the client and enters this information into the controller's Policy Table along with the client's MAC, input port, and violation. When revocation messages arrive from the Trusted Agent with the appropriate *keyID*, the Event Handler removes the associated client entry from the controller's Policy Table, and its security policy transitions to allow the client access to network privileges.

Policy Enforcer

The Policy Enforcer handles events passed to it from the Event Handler. It first confirms that arriving packets are not already flagged in the Policy Table. If not, the Policy En-

forcer next applies selected security policies against the arrived packet. If the packet passes specified checks, then it is passed to the Event Handler for normal forwarding. Otherwise, the Policy Enforcer returns *fields* and *ops* hash tables[3] to the Event Handler resulting in the client's traffic being redirected to the network's Trusted Agent. The Policy Enforcer also generates the randomized *passkey* and a unique *keyID*, which is passed back to the Event Handler with the client's other unique flow information (i.e., input port, MAC, and violation).

Policy Table

The Policy Table simply stores the identification and flag state information for each client. As we will later see in Fig. 5.5, the Policy Table stores *keyID* (primary identification key), *passkey* (for client authentication), MAC address, input port, and violation code for flagged clients (of which, minus the input port, are forwarded to the Trusted Agent).

5.4.2 Trusted Agent

The Trusted Agent serves as an intermediary between the client and network operator. For instance, the Trusted Agent is able to send revocation messages to the controller and reinstate the client's privileges in lieu of the network operator once the *passkey* is provided. It can also provide clients with instructions for regaining network access. Its components are shown in Fig. 5.2 and discussed next.

Client Policy Handler

The Client Policy Handler establishes a communication link with the controller to receive policy activation notices and submit revocation requests. When the Trusted Agent is first notified of a policy activation, it records the provided *keyID*, *passkey*, MAC, and violation associations in its Client Table as indicated in Fig. 5.5. The Client Policy Handler also pe-

[3]Hash tables are Ryuretic's method for directing network operations.

riodically queries the Revocation Table for *keyIDs* belonging to clients who have submitted a *passkey* and are awaiting the reinitialization of client privileges.

Client Table

The Client Table allows the Trusted Agent to maintain state for flagged clients. As shown in Fig. 5.5, this table maintains the client's *keyID*, *passkey*, MAC, and violation. It is also queried by the Client Table Handler to confirm client MAC and *passkey* pairs. Furthermore, the Client Table provides violation information to the Handler, so the Trusted Agent renders appropriate instructions to the client.

Revocation Table

The Revocation Table allows the Trusted Agent to queue the *keyIDs* of clients awaiting the reinstatement of their privileges. The Client Policy Handler then periodically queries the table for *keyIDs* and sends them in revocation messages to the controller.

Client Table Handler

The Client Table Handler queries the Client Table to verify the client's *passkey* and MAC address. When successful, the Handler loads the client's *keyID* to the Revocation Table where it is queued for delivery to the controller. As a security measure, the Client Table Handler can only query the Client Table and write to the Revocation Table.

Data Processor

The data processor is a Common Gateway Interface (CGI) module that provides server side scripting for the Trusted Agent's web server. It provides MAC and *passkey* associations to the Client Table Handler, and it renders feedback information to the client's web interface via HTML.

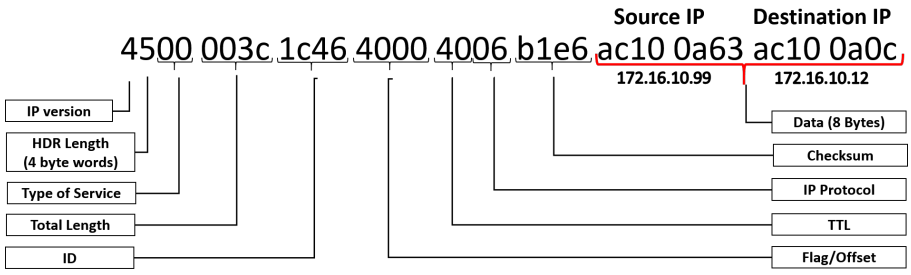


Figure 5.4: ICMP Packet Header

Web Server

While any number of web servers could be used for this component, this framework uses the lighttpd [62] web server due to its small memory footprint and support for Common Gateway Interface (CGI) scripts. The web server serves as the client's primary interface while resolving flags. It also captures the client's MAC address via a PHP script[4] when the client enters their passkey. The passkey and MAC address are then forwarded to the Data Processor for passkey validation.

5.4.3 Communication Channel

The SDN controller and Trusted Agent communicate with each other using ICMP to relay rule insertions, updates, and revocations. However, while the Trusted Agent can receive whole ICMP packets with complete payloads, the SDN controller only receives ICMP packet header information. This places a significant limitation on the amount of data that can be passed from the Trusted Agent to the controller. Consequently, the Trusted Agent must overwrite the 8-byte data field of the ICMP packet header, which normally contains the packet's source IP and destination IP (see Fig. 5.4), with its messages.

With limited space, the Trusted Agent constrains it responses to *action*, *keyID* strings. The *action* (see Table 5.1) value is a single letter abbreviation. It identifies the message type (i.e., initialize, acknowledge, update, or delete) sent by the Trusted Agent. For messages

[4]The PHP script for obtaining client MAC addresses requires that clients reside on the same subnet as the web server.

Table 5.1: Abbreviations Used for Controller Communication [30]

Abbr.	Meaning	Summary
i	initialize	Establish Trusted Agent Parameters
a	acknowledge	Send table entry receipt for keyID
u	update	Request update to refresh or replace current client table
d	delete	Request policy deletion for specified keyID

that do not require a *keyID*, (e.g., initialize) the *action* value is simply followed by a zero. For example, the Trusted Agent's initialization message to the Ryuretic controller appears as 'i,0'. However, messages from the controller have more flexibility. For instance, rule insertion methods destined for the Trusted Agent's Client Table will include MAC, *passkey*, violation, and *keyID* values. This format is recognized by the Trusted Agent and handled appropriately by its Client Policy Handler. It is through this communication channel and format that the Ryuretic controller and the Trusted agent are able to maintain corresponding tables (the Policy Table for the Ryuretic controller and the Client Table for the Trusted Agent), which are shown in Fig. 5.5. Moreover, while limited, this solution is easily adaptable to other SDN controllers using existing protocols.

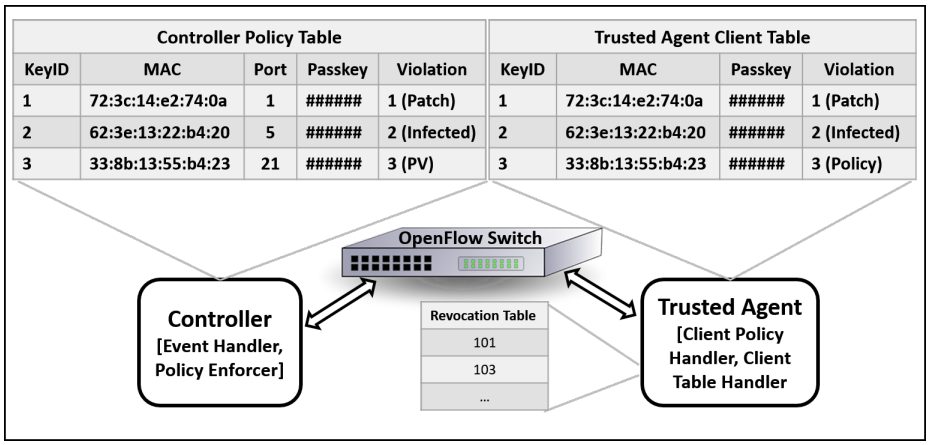


Figure 5.5: Controller - Trusted Agent Communication

5.5 Test Environment

We tested our framework using Mininet [24], which is a network emulator that creates virtual clients, switches, controllers, and links. All of our clients, including the Trusted Agent are virtual machines with Ubuntu 14.04 operating systems. The switch is OpenFlow 1.3 [12] capable, and Ryuretic applications run atop a RYU controller to provide network control. The testbed also provides Internet access via a virtual network address translator (NAT). Until a client is flagged, it is able to ping other client and access web services via the NAT device. Web services are tested using *curl* and *wget* commands and the Firefox Internet browser. However, when a client is flagged, its data flows are redirected to our Trusted Agent's web server until the client provides the appropriate *passkey*. Once that is provided, the client regains network privileges within 30 seconds of making the entry.

5.6 Example Use Case

To better demonstrate our security policy transition framework, we test our framework's ability to revoke a triggered security response for ARP spoofing. In doing so, we highlight key code listings contributing to the framework's detection and notification methods. We also discuss the transition process for security measures.

5.6.1 Spoofed ARP Packets

Spoofed ARP packets can poison the ARP tables of neighboring clients on a subnet and serve as a springboard for more dangerous attacks such as packet dropping (blackhole), rogue web server, and man-in-the-middle (MitM) attacks. Our framework allows the network operator to set a policy in Ryuretic to detect and trigger security measures for such attacks. When a packet arrives, Ryuretic determines its type and forwards it to the appropriate handler. For ARP, the packet is forwarded to the Event Handler's ARP handler (see Listing 5.1). The Event Handler first checks the incoming packet against its Policy Table.

If flagged, then the controller drops or responds to the packet in accordance with its ARP poison protocols. Due to space constraints, we do not include an example for this method[5].

Listing 5.1: Ryuretic ARP Event Handler

```
1     def handle_arp(self,pkt):
2         pkt_status = self.check_net_tbl(pkt['srcmac'], pkt['inport'])
3         if pkt_status is 'flagged':
4             fields,ops = self.Arp_Poison(pkt)
5         else:
6             spoofed = self.detectSpoof(pkt)
7         if spoofed:
8             self.notify_TA(pkt)
9             fields, ops = self.drop_ARP(pkt)
10        else:
11            fields, ops = self.respond_to_arp(pkt)
12        self.install_field_ops(pkt,fields,ops)
```

Listing 5.2: Ryuretic ARP Poison Detection Method

```
1     def detectSpoof(self,pkt):
2         policyFlag = False
3         if self.netView.has_key(pkt['inport']):
4             if pkt['srcmac']!= self.netView[pkt['inport']]['srcmac']:
5                 policyFlag = True
6             if pkt['srcip'] != self.netView[pkt['inport']]['srcip']:
7                 policyFlag = True
8         else:
9             self.netView[pkt['inport']] = {'srcmac': pkt['srcmac'],
10                                           'srcip': pkt['srcip']}
11        if policyFlag == True:
12            self.net_MacTbl[pkt['srcmac']] = {'stat':'flagged',
13                                              'port':pkt['inport']}
14            self.net_PortTbl[pkt['inport']] = {'stat': 'flagged'}
15        return policyFlag
```

If the source MAC or input port is not flagged, then the packet is passed to the detectSpoof() method (see Listing 5.2). If the packet is flagged as spoofed, then the Event Handler forwards a notification message to the Trusted Agent via ICMP. In turn, the Trusted Agent adds the flagged client to its Client Table. The message contains (MAC, *srcport*, *passkey*, violation, *keyID*) data values. Ryuretic's *fields* and *ops* hash tables are then set to match the packet and drop ARP replies from the sender. Otherwise, the packet is forwarded to the *respond_to_arp()* method for normal forwarding. As seen in Listing 5.2, the *detectSpoof()*

[5]See `https://github.com/Ryuretic/SecRev` for complete implementation.

method builds a network view that associates each client's MAC and IP address to a switch port. If a packet arrives after the network view is built with an incorrect MAC or IP address, then it is flagged for spoofing. Its network flows are now forwarded to the Trusted Agent, which renders a web page to inform the client of their violation and offer instructions for regaining access to the network (e.g., submit a user agreement policy, UAP). Currently, our system relies on the help desk to serve as the validating authority; however, future implementation could utilize patch software to provide compliance validation. Once the client obtains and submits the *passkey*, the client is then reinstated on the network and able to access its services–generally within 30 seconds of entering the *passkey*.

5.7 Discussion and Future Work

In future work, we hope to include additional fields in the Ryuretic programming framework to accomplish more robust communication channels for security policy transition. However, for the moment, our framework relies on current network protocols, while utilizing them in a disruptive way to reactively communicate the invocation and revocation of security responses. Doing so, allows us to automate these actions without network operator involvement.

While not demonstrated in this chapter, our work is also easily adaptable to a password-based authentication framework for clients seeking to join the network. In which case, the network view could be built as clients authenticate to the network. Likewise, with this framework, there is potential to build a host of readily available actions that clients can take once they are redirected to the Trusted Agent. For instance, the web portal can include patches, courses, administrative documents, initial warnings, etc.

By pushing these responsibilities to the Trusted Agent, this framework alleviates many day-to-day burdens experienced by network operators. Specifically, it eliminates the need for network operators to provide the additional configurations needed to reinstate a client's privileges on the network. It also provides network clients with immediate feedback con-

cerning their network privileges. Furthermore, this security framework is also applicable to cloud and virtual network environments. Using network functions virtualization (NFV) all components can be virtually created in a cloud infrastructure.

While SDN is capable of implementing numerous security features, this work does not assume that all security features will be handled by the controller. In fact, the introduction of our Trusted Agent further provides for the incorporation of additional security features where secondary devices serve to provide more layers to a defense in depth security strategy for detecting. Likewise, this work does not propose to replace application-level monitoring. Such services are still needed to identify a client's software version, provide patch compliance, detect malware, or even application-layer firewalling. However, the Trusted Agent could be configured to work with application-layer products in conjunction with amending security within the controller.

Network operators must also consider that more clients than just subscribers will operate on their networks (e.g., M2M communication or Web service interaction). If not handed appropriately, the redirection of flagged clients to a self-service interface, as proposed in this work, could cause IoT devices or user agents to assume the network has failed. Ideally, the network operator would *whitelist* or set aside specific ports for such devices to provide notifications if the device becomes flagged. For such cases, the Trusted Agent could also run a mail server to notify the help desk when a non-user device is affected.

5.8 Conclusion

With OpenFlow providing a vendor agnostic platform for SDNs and enabling the orchestration of numerous switches, programmers are better able to implement novel network applications for security and traffic engineering. Yet, network operators also need additional measures for automating daily processes to fully utilize SDNs in physical and virtual environments. We present a security policy transition framework that automates the flagging of clients, redirects their network flows to a Trusted Agent, and revokes activated security

measures once the client validates their network compliance and enters a *passkey* obtained from a validating authority (e.g., a help desk or patch software). This framework eliminates many daily network configuration requirements that network operators must now perform and improves the turnaround time for reinstating client access once compliance has been obtained. Turnaround times vary by violation and validation procedure, but reinstatement of network privileges occurs roughly 30 seconds after the correct *passkey* if entered.

5.9 Segue

This framework goes a long way towards reducing network involvement with network configurations. However, the revocation of policy enforcements is not the only configurations with which network operators need assistance. Many transition frameworks try to tie the network operator directly into the decision making process for detecting and denying malicious actors on their networks. Using this framework, the next chapter expands upon it to offer a means for the Trusted Agent to work with the SDN controller to perform active testing on clients. Within this process, clients move from flagged for testing, to deny or allow, and if denied, back to allow once network operator requirements are met.

CHAPTER 6

LEVERAGING SDN AND WEBRTC FOR ROGUE ACCESS POINT SECURITY

6.1 Introduction

Network security is a growing concern for government, industry, and campus networks, and hackers continuously test the limits of the security tools available to network operators. Rogue Access Points (RAPs), sometimes called malicious APs, such as wireless switches and wireless routers, are no exception [83, 84]. Generally speaking, RAPs are unauthorized wireless devices that, when connected to an organization's network, provide unauthorized wireless connectivity to other clients. While legitimate wireless access points (APs) give authorized clients greater mobility, help to accommodate the shortage of IPv4 addresses, and provide security or privacy for network clients, such devices can also allow hackers to bypass firewalls and intrusion detection prevention systems (IDPS). Other clients may use them to simply steal bandwidth.

Certainly, not all RAPs placed on a network are intentionally malicious. In some cases, clients do so to make their network access more convenient. However, clients often fail to enable even basic security features on these devices. As a result, hackers can access them via default settings and configure these devices to facilitate their own efforts. Researchers have also found that existing malware detection models are ill-equipped to deal with network address translation (NAT) devices (or RAPs) and report that roughly 19% of infected devices are hidden behind NATs [85]. For these reasons, we consider all RAPs to be a threat to an organization's network.

These devices also vary in functionality. Some RAPs act as simple switches, forwarding packets from various source addresses with no modification to the client packet's IP or MAC address. Throughout this chapter, such devices are referred to as wireless switches

(WS). Other rogue access points implement NAT, and we refer to these as wireless routers (WR) or wireless NATs. With wireless routers, one or many clients can reside on an unapproved subnet that obfuscates their MAC and IP from the rest of the network behind the RAP's public facing IP. Previous research has already established such devices as one of the most challenging problems facing network operators [86, 87, 88, 89].

While Ma et al. [90] identify four categories of RAPs (i.e., improperly configured AP, unauthorized AP, phishing AP, and compromised AP), our work only seeks to mitigate unauthorized APs by analyzing packets and traffic flows on the wire-side. As such, our work makes no attempt to recognize phishing APs since they may not actually be connected to the organization's network. Likewise, our work does not consider improperly configured or compromised APs since they are intentionally placed on the network by the network operator and are either hacked or exploited due to improper configuration. Unauthorized APs, however, are RAPs that have been placed on the network without the network operator's permission or knowledge. Henceforth, we will commonly refer to both unauthorized wireless switches and unauthorized wireless routers as RAPs.

To detect these devices, our framework, which we refer to as Network Flow Guard (NFG), offers a hybrid method, comprised of both passive and active measures, to detect and deny RAPs access to a network. Fortunately, Network Flow Guard's passive techniques are already capable of detecting the majority of available RAPs. For those not immediately detectable, Network Flow Guard passively monitors and analyzes client traffic signatures for RAP behavior. Suspected RAPs are then flagged, and Network Flow Guard temporarily redirects the suspected client's traffic to its *Trusted Agent* for active testing. Clients who fail the test are denied access to network services. Additionally, our active method introduces minimal burden to the network and can be applied in a random or rotational manner to periodically test all clients if preferred.

Beyond Network Flow Guard's focus on SDN-based, RAP detection and denial, our work also offers the following contributions:

- Establishes an initial framework for detecting and blocking rogue devices connected to SDN infrastructures
- Provides a public, open-source implementation of our work on GitHub [91]
- Provides a first use case for security emulation with Mininet-WiFi [25]
- Utilizes a combination of passive and active detection techniques to lessen the SDN controller's computation
- Uses the webRTC architecture to provide a novel NAT/RAP detection measure
- Improves features of Ryuretic [5, 91], a modular programming framework for SDN application development

This chapter is organized as follows. Section 6.2 provides background information concerning both NAT devices and RAPs. It also serves to motivate our reasoning for implementing RAP security along the edge-devices of an SDN. Section 6.3 discusses the current state of the art for detecting and preventing rogue devices (i.e., unauthorized NATs, wireless routers, and wireless switches) from accessing network resources. In Section 6.4, we discuss the assumptions and requirements for our design and the security features used in our work. Section 6.5 then describes the architecture and components comprising the Network Flow Guard architecture. We then discuss our test environment and test results in Section 6.6. In Section 6.7, we offer further discussion on Network Flow Guard's security implementation and its viability towards RAP security. Next, in Section 6.8, we discuss future work, and finally conclude in Section 6.9.

6.2 Background

Network security has emerged as a hot topic in industry, government, and campus organizations. Additionally, due to the security threats that rogue devices can pose (e.g., DHCP attacks, DoS attacks, data exfiltration, etc. [92]), Ma et al. [90] argue that detecting RAPs is one of IT departments' most important security functions. When physical port security is compromised, RAPs allow adversaries to commit espionage and launch attacks from outside the organization's perimeter. Moreover, it can be difficult to distinguish RAPs from

other clients or approved devices (e.g., VM servers, WAPs, NAT devices, etc.) when they also use a unique MAC-IP association.

6.2.1 Network Address Translation (NAT)

NAT devices act as an intermediary between private networks to other networks (e.g., the Internet). Because of NAT, a single IP address can represent an entire network. This feature has proven a valuable tool in overcoming shortages of IPv4 addresses. It also allows network operators to partition their networks into smaller and more manageable networks to hide network topology, provide client anonymity, and provide content filtering. However, these devices introduce new vulnerabilities to networks as well.

Since it is possible to hide one or more networks behind a single IP address, clients connected to these rogue devices can gain unrestricted access to the network. This threat is particularly concerning with wireless NAT devices (e.g., wireless routers). Additionally, while traffic generated by a NAT is an amalgamation of the traffic generated by its connected clients, it can still be difficult to detect due to regular network traffic providing concealment.

6.2.2 Rogue Access Points (RAP)

A RAP is essentially an unauthorized AP connected to a campus, industry, or government network by unaware or malicious insiders. Unfortunately, multiple studies indicate that nearly 20% of all corporations have active RAPs in their network at some time [92, 93, 94, 84]. And, as we have previously discussed, these devices may act as wireless switches or wireless routers.

For such devices, Beyah et al. [95] identify two undesirable outcomes: 1) an employee installs a RAP for convenience and a malicious user finds the unsecured connection via *wardriving*; or 2) a hacker connects a RAP to a live port within an organization and uses that device to run exploits from outside the organization's security perimeter. These de-

vices can include wireless routers, wireless (MAC-learning) switches, or modified clients. Once installed, any number of exploits are available to the attacker, such as additional hacking, sniffing wireless traffic, or performing man-in-the-middle (MitM) attacks [84]. RAPs can also pose a greater threat to local networks since they allow attackers to run exploits from outside the security of the building's network architecture. As a result, firewalls, port blocking (e.g., 802.1x and NAC), anti-virus, WPA2 encryption or wire side scanners are circumvented [92].

6.3 Rogue Device Detection Methods

Due to the potential threat RAPs pose, numerous studies in recent years have sought to detect and prevent RAPs on traditional networks. Developed techniques include client-side (wireless-side), server-side (wire-side), and hybrid approaches [93]. However, since our solution only captures traffic via an OpenFlow-enabled switch, a server-side approach, our work only touches upon client-side solutions and avoids discussion of hybrid solutions altogether while addressing the server-side approaches most relevant to our work. For completeness, Table 6.1 provides an overview of the many detection methods considered in our research.

6.3.1 Client-side approach

Some client-side solutions use sensor-based wireless intrusion prevention systems (WIPS) to identify, locate, and block RAPs. Aruba Networks, AirMagnet, and AirDefense represent a few vendors offering such devices. However, the equipment can be cost prohibitive, may not cover all required areas, and may have legal implications. For instance, placing WiFi units in *rogue-AP* blocking mode, instead of just detecting, can interfere with legitimate signals (e.g., a phone or carrier *hotspot* signal) [122]. As a result, some technologies for blocking RAPs have led to heavy fines by the FCC [122]. Likewise, security protocols like WPA2 can prevent unauthorized devices from associating with a legitimate wireless

Table 6.1: Server Side NAT Detection Techniques

Technique	NAT	RAP	ID	Approach	Method	Issues
Inter-Packet Spacing [95, 96, 97]	U	Yes	U	Passive	Observe traffic leaving and entering the network over time to detect patterns in inter-packet spacing	Requires sniffer and port mirror or tap
Inter-Packet Arrival [98, 99]	U	Yes	U	Passive	Observe traffic arrival times and create signature for delay between packets	Requires sniffer and port mirror or tap
TTL Dec.[100, 101, 102, 103, 104]	Yes	Yes	No	Passive	Monitoring for TTL decrementing	Devices may not decrement TTL
TCP-ACK pairs [96]	Yes	Yes	No	Passive	Detects longer time intervals between two consecutively sent packets, such as TCP ACKs	Packets may be queued by a busy router, skewing results
TCP-SYN [103]	Yes	Yes	No	Passive	Monitors the various lengths of TCP SYN packets	Use of same OS
Multi-OS [102]	Yes	Yes	No	Passive	Mapping TTLs to a port to detect multiple OSes at that port	Multiple devices could use the same OS (same TTLs)
IP ID [105]	Yes	Yes	No	Passive	Observes patterns in IP ID sequences to detect multiple clients	Enabling Don't Fragment (DF) bit and random IP ID generation
OS-Finger printing [106, 107, 102, 108]	Yes	Yes	No	Passive	Uses IP ID, TCP SEQ number, TCP SRC port, TTL, window size, etc. to ID multi-OS	Won't detect OS like OpenBSD due to random field data
Wireless-Finger printing [109]	No	Yes	No	Passive	Uses DCF and rate adaptation specifications to ID wireless devices	DCF manipulation, port mirror, additional equipment
Application-Type [102, 110]	Yes	Yes	No	Passive	Maps applications (e.g. Firefox) to port using HTTP User-Agent data	Multiple devices could use the same application
Multi-MAC detection	Yes	Yes	No	Passive	Detects multiple MAC addresses on one port	Devices can be bridged with multiple IPs, but share a single MAC
Device MAC	Yes	Yes	No	Passive	Different vendors are known to use unique values in MAC addresses	MAC addresses could be spoofed
RTT [111, 93]	Yes	Yes	No	Passive	Measures RTT from host to DNS or other server	Limited accuracy
TCP/IP packet payloads [110]	Yes	Yes	No	Passive	Monitors Cookies contained in payloads	Processor intensive and cookies can be disabled
HTTP logs [110]	Yes	Yes	No	Passive	Uses that clients make HTTP requests–one of the most used protocols	Requires training and processing of server logs
Port Monitoring [112]	Yes	Yes	No	Passive	NAT devices often associate high number ports with hidden IPs	Not a certainty
Clock Skew [113, 114, 115]	Yes	Yes	No	Passive	Uses clock skews created by computer chips to ID devices	TCP time stamp can be disabled or altered
NetFlow Analysis [113, 103, 116]	Yes	Yes	No	Passive	Monitors dst and src data for MAC, IP and port. Uses machine learning techniques (SVM, C4.5)	Probability of false positives, Training requirements
Radio Freq. [117]	No	Yes	No	Passive	Sniffs beacons, SSID, etc. from radio waves	Expensive, inefficient, poor coverage
CSBBSW [88]	Yes	Yes	U	Passive	Maintains running client-side bottleneck BW to detect RAPs	Assumes more than one host connected to RAP
TCP Timestamps [118, 119]	Yes	Yes	No	Passive	Compares system boot time with TCP timestamp to detect multiple clients	Non-constant errors (jitter) decreases matching quality
NMAP [120]	Yes	Yes	No	Active	Sends active packets to probe client ports	Can alert hackers to detection measures; additional traffic to network
Localization [117, 121]	No	Yes	Yes	Active	WLAN localization algorithms can achieve accuracy of 35 m	Often require extensive manual profiling to build RF maps

RAP includes wireless routers and switches. ID field indicates if method distinguishes between NAT and RAP. U - Undetermined

APs, but these security controls are only enforced by APs managed by network operators [92]. They do little for rogue devices that connect directly to an Ethernet port. Additionally, these rogue devices operate at a layer below anti-virus and intrusion detection prevention systems (IDPS), rendering their protection ineffective [92]. Even with 802.1x port control, which most networks don't have, RAP configurations are still difficult to completely circumvent. For instance, a RAP could obtain network access through a bridging laptop [92]. Other solutions, like having client devices monitor round-trip time (RTT) [111] or received signal strength (RSS) and then report findings to a local server, require software to be installed on client devices and may not be acceptable to *bring your own device* (BYOD) initiatives.

Network operators may also use handheld wireless detectors [123, 124] requiring them to physically search for RAPs. This is far from an ideal solution, since multiple frequencies may be used and the area may be too large for an operator to quickly cover. Moreover, as [125] observes, if savvy users employ directional antennas or other signal dampening techniques, detecting these devices may not be possible. An additional fallacy of client-side scanners is that they also falsely flag local hotspots, which can include smartphones, wireless-enabled vehicles, neighboring business APs, and residential APs. However, while there is still room to enhance client-side approaches to detect rogue devices, such efforts are outside the scope of this work and not considered beyond this summary.

6.3.2 Server-side approach

Within server-side or wire-side solutions, three approaches contribute to our work's overall design to detect and deny RAP access to organizational networks. These include passive detection, active detection, and SDN capabilities.

Passive Detection

Passive approaches to detecting RAPs are often hard for attackers to detect, and they are fairly easy to implement. The passive approach involves monitoring network traffic and inferring outcomes based on those observations. Often the observation point is a port mirror or LAN tap located somewhere in the network where the attacker's traffic traverses. Various methods use the time-to-live (TTL) field in the IPv4 header to detect NAT traffic [100, 101, 103, 104]. Maier et al. [104] also consider user-agent strings from HTTP requests along with the TTL field of IP packet headers to passively identify NAT devices. Their work observes that the initial TTL for Windows, MacOS, and Debian-based systems have initial values of 128, 64, and 64, respectively. They also observe that NAT devices typically decrement the TTL field before relaying a packet [102]. So, if a decrement is detected while monitoring a switch's port mirror, then the device is flagged as a NAT. They also implement a method for flagging devices when multiple TTLs are found to originate from a single port [104]. Unfortunately, problems with these methods include the need for middleboxes to sniff traffic (which introduces financial, power, and space costs). Also, a lack of knowledge regarding the monitoring port's location in regard to the rogue device can make TTL values difficult to intuit. [104] also use user-agent strings to detect RAPs; however, this method is completely thwarted by HTTPS encryption and is not considered in our work.

Other solutions involve analyzing Ethernet headers for company MAC address assignments [126]. Of course, if a rogue device behaves as a switch, then the detection of multiple MAC or IP addresses is also a strong indicator. However, Beyah et al. argue that rogue devices with MAC address spoofing and NAT capabilities can easily circumvent current detection methods [95]. Instead, they observe that temporal traffic characteristics (e.g., link speed, wireless link capacity, and traffic patterns), which are unique to wireless APs, can be applied to correlate inter-packet space arrivals of FTP packets with RAPs. Their method is able to detect RAPs with 90% accuracy and a false positive rate of 20%. However, their work, much like Maier et al., also struggles with placement of the monitoring

port and switch load. For instance, inter-packet spacing methods can lose accuracy as more switches are placed between the RAP and the monitoring device.

Wei et al. [96] implement two algorithms that perform sequential hypothesis tests to evaluate the inter-ACK arrival times of TCP headers collected at a monitoring port. To differentiate between wireless and wired connections, these algorithms exploit properties inherent in 802.11 CSMA/CA and the half duplex nature of wireless channels. Kao et al. [88] also use inter-packet arrival times to calculate the average bandwidth to detect wireless clients. Their method offers an accuracy of 99%, and detected APs that are not *whitelisted* are tagged as RAPs.

Gokcen et al. [102] look for patterns in network traffic to footprint NAT-like behaviors and propose a passive machine learning approach using the C4.5 machine learning algorithm. Using 41 classifying features available in NetMate, an open source flow generator [127], they fingerprint NAT behavior. Yet, machine learning algorithms require training and large datasets that may vary from one network to the next, thus limiting its portability. Hence, our work avoids such approaches. Komárek [110] introduces a server-side solution that is solely interested in detecting whether or not a given IP address found in proxy logs is actually an end host. Komárek also assumes that a NAT device will be more active and exhibit a mix of client behaviors when compared to single clients. However, a malicious client may set up a wireless NAT for their sole use, so a mixture of behaviors would go undetected. In contrast to the above approaches, Venkataraman et al. [109] make no assumptions regarding whether wired networks are faster than wireless. Instead, they focus on 802.11 MAC layer features, such as the Distributed Coordination Function (DCF) and rate adaptation specifications, to fingerprint wireless traffic using a Bayesian classifier with 86-90% accuracy [109].

Active detection

Since, many passive solutions are only capable of achieving detection rates of 85-97%, our work also considers active techniques for rogue device detection. For instance, many RAPs maintain open ports for HTTP requests, allowing clients to log into the system remotely and configure the device. Other ports may allow SSH connectivity. If these ports are found to be open by a scanner such as NMAP [128], then one might conclude that a RAP exists. However, active detectors, like NMAP [128] can be slow, intrusive, and noisy when used to scan indiscriminately. They also do not audit access points, and they can alert clients that active scans are occurring by triggering personal firewall and IDS alerts [126]. Additionally, some RAP devices may be configured to only open ports when granting one of its clients access to the network. Resultantly, NMAP results may not indicate the presence of a RAP.

For the above reasons, our work avoids the active scanning approach and instead attempts to detect wireless NATs using a technique we refer to as *port intercepting*. This method essentially seeks to intercept a newly opened port's traffic flow during a TCP communication stand-up. Once intercepted, traffic flows are redirected to our *Trusted Agent* where our active measures are deployed. However, since this feature represents one of the contributions of our work, further discussion is postponed until Section 6.4.

SDN Approach

Software-defined networks (SDNs) separate the control plan from the data plane to allow network operators to orchestrate network activities from a logically centralized controller.[1] Accordingly, SDN is attracting noticeable attention, and both industry and academia are now deploying SDNs. In addition, many researchers have already demonstrated that SDN can be used to thwart ARP Poisoning [70, 34], DDOS [129, 130], Rogue DHCP servers [131, 33], etc. without the use of excess equipment and on current network infrastructures.

[1] Controllers can still be physically distributed

However, to our knowledge, no work has been published on detecting and preventing RAPs from gaining access to SDNs. Hence, our work seeks to add to this growing body of literature.

Fortunately, many of the passive techniques previously discussed are already fully implementable by SDN, where OpenFlow-enabled[2] switches serve as the network's primary detection device. This makes SDN a powerful detection mechanism, since OpenFlow switches have unique access to state and header information located at the network's edge. For instance, the SDN switch is able to double as a monitor. Thus our work benefits from a known monitoring point (the client's point of entry). This monitoring also offers a new level of control for the network. Furthermore, as mentioned previously, it represents a first attempt at utilizing any of the previously mentioned detection methods in conjunction with the dynamic capabilities of SDNs to detect and remove RAPs.

6.4 Network Flow Guard Security

In this section, we discuss the goals, design, and components for our Network Flow Guard[3] security implementation. In doing so, we show how an OpenFlow switch, linked to a Ryu [32] SDN controller, using Ryuretic [5], and working in cooperation with a *Trusted Agent*, can serve as the primary intrusion detection/prevention system (IDPS) for an SDN.

6.4.1 Requirements and Assumptions

An essential requirement of our work is to distinguish rogue devices from legitimate wireless APs. However, since rogue devices can also include wired NAT devices that present some of the same challenges as RAPs, we do not find it necessary to distinguish whether the device is a NAT or a RAP in our work. Hence, identifying the presence and blocking access of either device is considered acceptable in our work. Further, we seek to do so with minimal impact to network performance.

[2]Other southbound interfaces exist; however, OpenFlow is still considered the defacto standard [42]
[3]Network Flow Guard (NFG) was first introduced in Chap. 2 and Chap. 3.

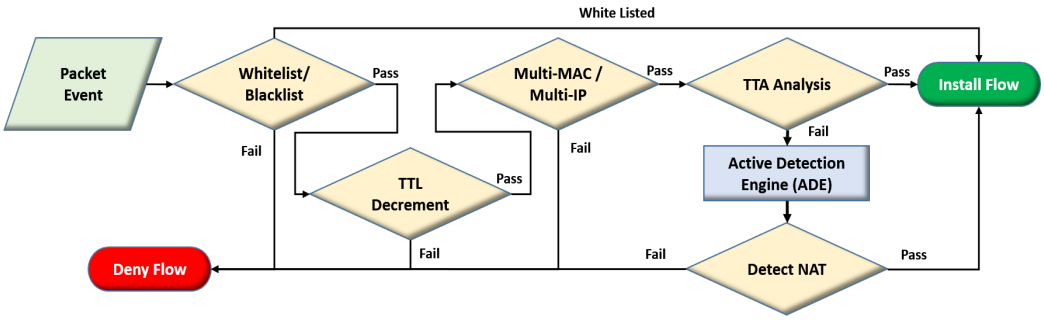


Figure 6.1: Network Flow Guard's RAP Detection Flowchart

Our work also makes some basic assumptions. For instance, we assume that multiple MACs or IPs are visible through bridged devices, while wireless routers and NATs only provide a single facing MAC and IP. We also assume that clients are directly connected to an OpenFlow switch, and that legitimate wireless APs can be *whitelisted*. Additionally, our work also considers the unique characteristics of operating systems, web browsers, and network traffic (i.e., extra hop traffic characteristics created by a RAP insertion) along with the number of MAC and IP addresses assigned to each port.

6.4.2 Security Features

Our solution employs several methods for detecting rogue devices. As shown in Fig. 6.1, our algorithm initially passively inspect packets against a *whitelist/blacklist* table before looking for TTL and multiple MACs transiting a single port. As it happens, these methods alone can defeat the majority of rogue devices and, hence thwart novice attackers. In fact, as shown in Table 6.2 where we provide the results of our observations for both wireless routers (WR) and wireless switches (WS), we find that TTL decrement and multi-mac (or multi-IP) detection are sufficient to detect all of the devices tested during our research, with the exception of one scenario. Should a wireless switch have only one client, then multiple MACs will not be detected and no TTL decrement will occur. However, our work also considers other devices that implement a NAT but are configured to not decrement the packet TTLs. For these, we implement a passive flagging scheme and an active detection

Table 6.2: RAP Characteristics by Type

Wireless AP	Type	IP	MAC	NAT	TTL Decr.	Multi Mac/IP
CiscoLinksis E2000	WR	Yes	Yes	Yes	Yes	No
ASUS RT-N56U	WR	Yes	Yes	Yes	Yes	No
NetGear N300	WR	Yes	Yes	Yes	Yes	No
NetGear N600	WR	Yes	Yes	Yes	Yes	No
Buffalo WZR-HP-G300NH	WR	Yes	Yes	Yes	Yes	No
ZyXEL	WS	No	No	No	No	Yes
TP-Link	WS	No	No	No	No	Yes
TrendNet TEW-430APB	WS	No	No	No	No	Yes

measure.

Consequently, since active methods for detecting RAPs add extra congestion to networks and are detectable by attackers, our work uses a hybrid approach as shown in Fig. 6.1. These passive detection methods identify RAPs using three algorithms: TTL check, Multi-MAC/IP, and TCP time-to-ack analysis. Ports that are flagged for TTL decrement or multi-MAC/IP are immediately denied access to network services, meaning all traffic is either redirected to the *Trusted Agent* or dropped. On the other hand, ports flagged by the time-to-ack (TTA) analyzer are redirected to Network Flow Guard's *Trusted Agent* for active detection. The Network Flow Guard architecture uses the following components.

Whitelist/Blacklist

Many NAT devices and wireless APs are intentionally used by network operators. To avoid flagging approved devices or wasting valuable compute, Network Flow Guard exempts *whitelisted* packets having the appropriate MAC-IP-port association. Of course, any detected wireless AP or NAT not listed on the *whitelist* is flagged as a RAP. Network Flow Guard then reconfigures the OpenFlow switch to redirect packets from the device to its *Trusted Agent*.

TTL Check

The time-to-live (TTL) checker serves as the next line of defense in the Network Flow Guard framework and is completely handled by the SDN controller. When an IP packet is detected, Network Flow Guard checks its TTL value in the packet's IP header, which should be either 64 or 128. If the TTL is ttl_{init}-1 or some other value, then the device is flagged as a rogue device. Ports flagged by the TTL checker are immediately blocked via the SDN controller's 'deny' policy. To confirm this method's effectiveness, we tested several wireless routers (WR) and wireless switches (WS). The results[4] are shown in Table 6.2. In all of the wireless routers we tested, we observe that TTL decrement does occur; however, as discussed in Section 6.1, wireless switches simply forward packets without modification. As a result, they go undetected by this method, since they do not decrement the packet's TTL. Further, we observe that some wireless routers (e.g., the NetGear N300) only consistently decrement the TTLs of TCP[FIN, ACK], TCP[RST, ACK] and QUIC packets. It also decrements the TTL of the first TCP[SYN] and TCP[ACK] packets. In this case, we conclude that the wireless router is relying on its client to establish communication, but to improve performance, it handles the rest of the TCP communication until it receives a TCP[FIN,ACK] or TCP[RST, ACK]. Even so, any decrementing makes this wireless router detectable by Network Flow Guard.

Multi-MAC/IP Association

This algorithm simply maintains a running count of MAC and IP addresses associated with each port of the switch. If the number of MAC addresses or IP addresses exceed one within a given time span, we choose 60 seconds, then the single host assumption is violated. This algorithm is primarily in place to catch RAPs that might bridge their clients through to the network. This method works for the wireless switches (WS) shown in Table 6.2 with one exception. Should the wireless switch only have one client, then this method fails to detect

[4]See https://github.com/Ryuretic/WiresharkCaptures for Wireshark captures.

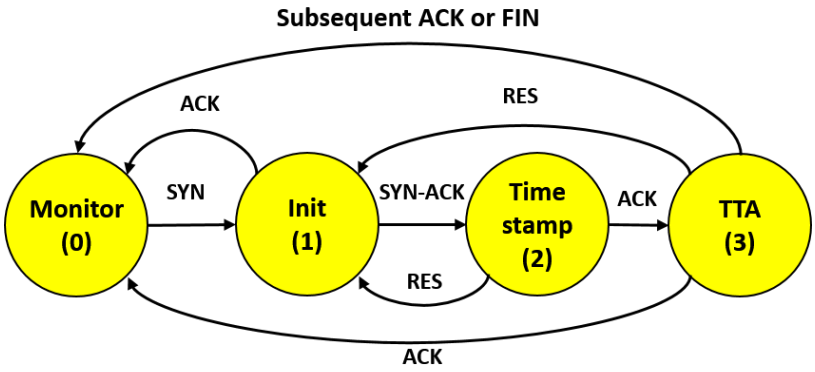


Figure 6.2: State Diagram for Capture of Port Flow TTA

its presence; however, for those that are detected via this method, they are automatically added to the controllers' 'deny' policy.

Time-to-Acknowledge (TTA) Analysis

One method of detecting RAPs includes monitoring the time it takes for a host to acknowledge received communications. By tracking the round-trip time (RTT) of the three-way handshakes of TCP connection establishments for individual IP addresses, [132] observe that 100% of wired packets have an RTT of less than 1ms, while 80% of wireless packets have an RTT exceeding 1ms. They also observe that they are able to successfully detect wireless nodes with only 5% of RTT values. We adapt these findings to our own work; however, since SDN allows the switch to serve as our monitor, we calculate the time-to-ack (TTA) at each switch port rather than by IP address (which could be spoofed).

To do this, Network Flow Guard's time-to-ack analysis module maintains a running average for each switch port by capturing the time-to-ack (time between a destination IP's TCP[SYN,ACK] packet and the client's TCP[ACK] packet) and adds it to the switch port's running average (*portAv*). Hence, Network Flow Guard must also maintain a state table for the TCP connections of all client-connected switch ports. A state transition diagram for a single time-to-ack capture is shown in Fig. 6.2. Here, the controller monitors for TCP[SYN] packets originating from a client before initializing an entry in its state table.

Once initiated, the controller monitors incoming TCP[SYN,ACK] packets destined for the client's corresponding IP and source port. When observed, it records a timestamp[5] for the packet's arrival. Next, the controller monitors for the corresponding client's TCP[ACK] packet and records its timestamp. The difference between timestamps is recorded as a time-to-ack. Network Flow Guard then removes the table entry for the TCP connection.

Once the time-to-ack is obtained, the value is entered into either Eqn. 6.1 or Eqn. 6.2 depending on whether the port average (*portAv*) is being initialized or not. Eqn. 6.1 calculates an initial port average after the controller obtains its first time-to-ack. This value, seeded with an initial value of 5ms (see Section 6.6), starts a running average. Eqn. 6.2 calculates subsequent port averages for the switch port. In both equations, either the seed or the old *portAv* are weighted with a factor of 9 to ensure gradual changes (see Section 6.6).

$$portAv_{init} = \frac{(weight * seed) + TTA}{(weight + 1)} \tag{6.1}$$

$$portAv_{next} = \frac{(weight * portAv_{old}) + TTA}{(weight + 1)} \tag{6.2}$$

Since this method only requires the first three packets of a TCP handshake (i.e., TCP[SYN], TCP[SYN,ACK], TCP[ACK]), Network Flow Guard shunts remaining packets from being passed to the controller by installing a temporary *Match-Action* rule to the switch once the TCP[ACK] is observed. This rule allows remaining packets to traverse the switch without controller involvement until the flow completes. Moreover, eliminating the extra packet load to the controller leaves additional compute available for other processing needs. Finally, the *portAv* for each port is compared with that of the other switch ports, and ports having the highest averages are flagged for active testing.

[5]Ryuretic provides a timestamp for each packet object it creates.

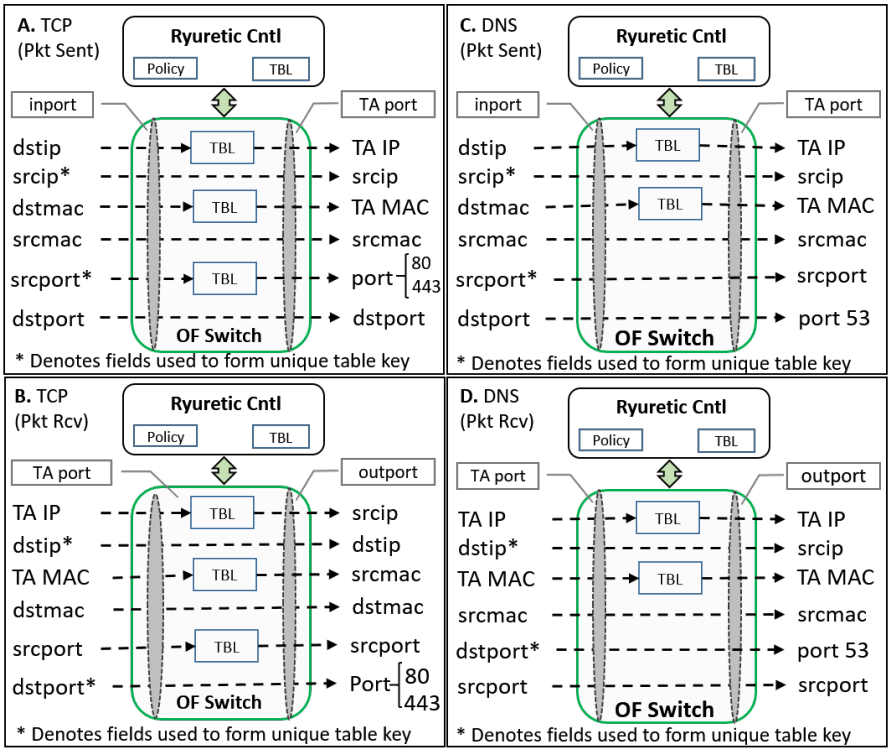


Figure 6.3: Port Interception on SDN Switch

Active Detection Engine

For Network Flow Guard, there are actually two options for utilizing the active detection engine (ADE). In one case, the ADE is utilized anytime a port is flagged as a possible NAT by time-to-ack analysis. In the other case, the ADE is used against all hosts on a random, rotational basis. When a client is flagged (or selected), the controller dynamically redirects HTTP(S) requests to Network Flow Guard's *Trusted Agent*, which delivers a PHP/JavaScript encoded index.php file to the suspected host and renders an HTML5 form back to the *Trusted Agent*. If flagged by the ADE, then the controller updates its policy table to 'deny' the client's access to the network.

The technique Network Flow Guard uses to deliver its index.php file is one we refer to as *port intercepting*. Since Network Flow Guard can only deliver its index.php file through ports that are opened by the client's browser, a new feature is added to the Ryuretic framework (see Section 6.5) to allow the controller to intercept DNS and TCP traffic destined

for one IP address and redirect it to Network Flow Guard's *Trusted Agent*. This new feature further allows Network Flow Guard to modify packet header information as well. As shown in Fig. 6.3, when Network Flow Guard's policy table indicates that a client's *inport*, MAC, or IP is flagged for 'deny' or 'test', it redirects the client's DNS and TCP traffic to the *Trusted Agent*. To do so, the SDN controller maintains the flow's intended destination MAC, IP, and Port (for TCP only) under a unique *keyID* using the client's *srcip* and *srcport*. Hence, when the *Trusted Agent* responds, the controller completes a backwards lookup and again reinserts the packet's original values to complete the TCP or DNS communication flow. As a result, the client sees the *Trusted Agent*'s DNS server and web server as its intended destination. Moreover, with the client's DNS and TCP being intercepted by the controller and redirected to the *Trusted Agent*, Network Flow Guard is able to incorporate its novel detection method.

Network Flow Guard's active detection engine (ADE) takes advantage of an emerging application program interface (API) for real-time communication, webRTC, which is now natively compatible with Google Chrome, Opera and Firefox web browsers [133, 134] and having plugins for Internet Explorer and Safari. Moreover, more than one billion endpoints devices are already utilizing a webRTC-enabled browser [133]. WebRTC, which is intended to provide real-time, low cost, high quality audio and video, and data communication between network devices [133], can also be used to render a client's local IP address and ports via NAT traversal features enabled by its RTCPeerConnection API. The RTCPeerConnection API is intended to allow two peers to communicate directly, browser to browser [134]. Utilizing this API, Network Flow Guard's *Trusted Agent* renders a PHP/JavaScript encoded index.php file to the client's browser that seeks to identify subnets.

PHP, being a server-side programming language captures the client's IP address as seen by the *Trusted Agent*. However, the JavaScript language is a client-side programming language, and it runs webRTC's RTCPeerConnection API to initiate a hidden communication with the client's browser, which allows our method to obtain the client's local IP address.

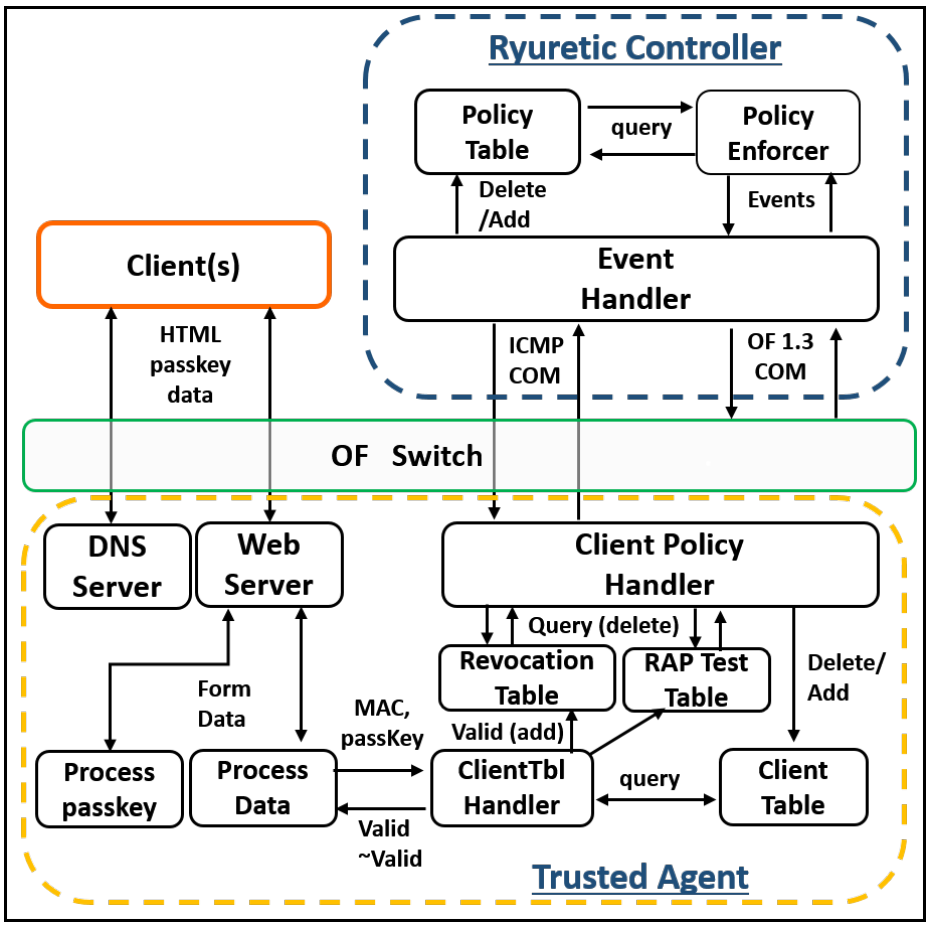


Figure 6.4: NFG RAP Detection Framework

Both values are returned to the *Trusted Agent* via a hypertext markup language (HTML5) post form method. Once received, the values are compared using a Common Gateway Interface (CGI) module (i.e. the Process Data module in Fig. 6.4) that provides server-side scripting for the *Trusted Agent*'s web server. As of this work, we believe this to be the first use of webRTC to actually detect the presence of subnets hiding behind NAT devices, making it a disruptive and novel contribution to the body of literature for detecting RAPs.

6.5 Network Flow Guard Architecture

For this project, we use Ryuretic [5, 91], which is a domain specific language (DSL) offering a modular framework for application development atop the Ryu [32] controller. Ryuretic [5, 91] provides a means for network operators to rapidly prototype and exper-

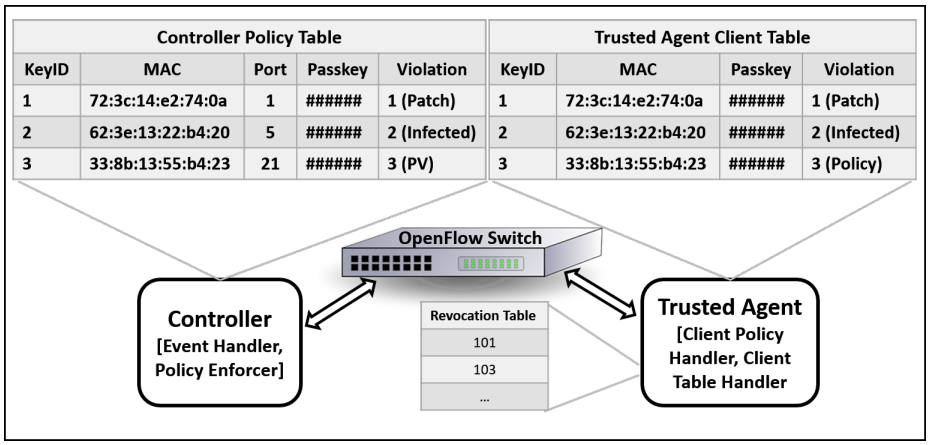


Figure 6.5: Controller - *Trusted Agent* State Information

imentally evaluate future routing or security applications for either wired or wireless environments. It also provides an intuitively simple format for network operators to select header fields within a packet (pkt[*]) and then specify what operation (*ops[*]*) occurs when a match (*fields[*]*) is found [6]. We specifically developed this language for modular application development on Ryu controllers using OpenFlow 1.3, which allows access to the header fields like the TTL. Further explanation is provided in Section 6.7.

Another feature of this architecture comes from our earlier work [30] where we develop a framework for handling security policy transitions within SDNs. In [30] work, ports are not simply blocked, but referred to a *Trusted Agent* when the port's attached client is flagged for violating one of the network operator's security policies. That framework is further expanded in our work to support Network Flow Guard's active detection measure. In [30], we develop an ICMP-based communication (an in-band channel) protocol to send information from the controller to its *Trusted Agent*, and vice versa. This communication channel is used to submit security policy amendments; however, we expand this protocol to communicate active test results in this work. Additionally, this communication allows both the controller and the *Trusted Agent* to maintain corresponding state tables as shown in Fig. 6.5. Next, we discuss the components comprising Network Flow Guard, as shown in Fig. 6.4, in the following subsections.

[6]The * symbol is used to represent the respective key within the objects fields and ops.

6.5.1 SDN Controller

While Network Flow Guard uses a Ryu [32] SDN controller supporting OpenFlow 1.3 [42], its applications are written in Ryuretic [91, 5]). These modules consist of the Event Handler, Policy Enforcer, and Policy Table. These modules are next discussed.

Event Handler

The Event Handler serves as the primary interface for the controller, responding to network events from the switch, security events from the Policy Enforcer, security policy transitions, and test notifications from the *Trusted Agent*. It also handles insert and delete messages for the controller's Policy Table to maintain state for each connected client. Once a client is flagged, the Event Handler creates a unique *passkey* and a *keyID* for the client and enters this information into the controller's Policy Table along with the client's MAC, input port, and flag as shown in Fig. 6.5. When revocation messages arrive from the *Trusted Agent* with the appropriate *keyID*, the Event Handler removes the associated client entry from the controller's Policy Table (or updates the flag) and amends its security policy to allow (or officially 'deny') the client's access to network resources.

Policy Enforcer

The Policy Enforcer handles events passed to it from the Event Handler. It first confirms that arriving packets are not already flagged in the Policy Table. If not, the Policy Enforcer next applies selected security policies against the arrived packet. If the packet passes specified checks, then it is passed to the Event Handler for normal forwarding. Otherwise, the Policy Enforcer returns *fields* and *ops* hash tables[7] to the Event Handler resulting in the client's traffic being redirected to Network Flow Guard's *Trusted Agent*. This also occurs if a security policy flags the client for active testing. The Policy Enforcer also generates the randomized *passkey* and a unique *keyID*, which is passed back to the Event Handler with

[7]Hash tables are Ryuretic's method for directing network operations.

the client's other unique flow information (i.e., input port, MAC, and flag).

Policy Table

The Policy Table simply stores the identification and flag state information for each client. As shown earlier in Fig. 6.5, the Policy Table stores *keyID* (primary identification key), *passkey* (for client authentication), MAC address, input port, and flag for flagged clients (all of which are forwarded to the *Trusted Agent*).

6.5.2 Trusted Agent

The *Trusted Agent* serves as an intermediary between the client and the network operator. For instance, the *Trusted Agent* is able to send revocation messages to the controller and reinstate the client's privileges in lieu of the network operator once the *passkey* is provided. It can also provide clients with instructions for regaining network access. Its components are shown in Fig. 6.4 and discussed next. Additionally, as of this work, the *Trusted Agent* also performs active testing on behalf of the controller, and notifies whether the client's status should be reinstated or denied based on test results.

Client Policy Handler

The Client Policy Handler establishes a communication link with the controller to receive policy activation notices and submit revocation requests. When the *Trusted Agent* is first notified of a policy activation, it records the provided *keyID*, *passkey*, MAC, and violation associations in its Client Table as indicated in Fig. 6.4 and Fig. 6.5. The Client Policy Handler also periodically queries the Revocation Table for *keyIDs* belonging to clients who submit a *passkey* in order to regain client privileges. As of this work, the Client Policy Handler also queries the RAP test table (see Section 6.5.2) for *keyIDs* and test results to provide results back to the controller.

Client Table

The Client Table allows the *Trusted Agent* to maintain state for flagged clients. As shown in Fig. 6.5, this table maintains the client's *keyID*, *passkey*, MAC, and flag. It is also queried by the Client Table Handler to confirm client MAC and *passkey* pairs. Furthermore, the Client Table provides violation information to the Handler, so the *Trusted Agent* renders appropriate instructions to the client.

Revocation Table

The Revocation Table allows the *Trusted Agent* to queue the *keyIDs* of clients awaiting the reinstatement of their privileges. The Client Policy Handler then periodically queries the table for *keyIDs* and sends them in revocation messages to the controller.

RAP Test Table

The RAP Test Table allows the *Trusted Agent* to queue the *keyIDs* and test results of clients currently under 'test'. The results are a simple 'pass' or 'fail' value. The Client Policy Handler then periodically queries the table for *keyIDs* and *results* and sends them in update messages to the SDN controller.

Client Table Handler

The Client Table Handler queries the Client Table to verify the client's *passkey* and MAC address or the client's active test data. When successful, the Client Table Handler loads the client's *keyID* to the Revocation Table (or the client's *keyID* and *result* to the RAP Test Table) where it is queued for delivery to the controller. As a security measure, the Client Table Handler can only query the Client Table and write to the Revocation Table or RAP Test Table.

Table 6.3: Controller Communication Abbreviations

Abbr.	Meaning	Summary
i	Initialize	Establish Trusted Agent Parameters
a	Acknowledge	Send table entry receipt for keyID
d	Delete	Request policy deletion for specified keyID
r	Result	Send test result notice

Data and Passkey Processor

The *data* and *passkey* processor consists of two Common Gateway Interface modules that provide server side scripting for the *Trusted Agent*'s web server. It provides MAC and *passkey* associations to the Client Table Handler, and it also evaluates form data it receives from the client's browser to determine if the client is situated behind a NAT device. It then renders feedback information to the client's web interface via HTML.

DNS Server

The DNS server provides a response to DNS queries receives as a result of a client's port being intercepted. In this project, we use a Bind9 DNS server and configure it to render our *Trusted Agent*'s IP address as the address for any DNS query it receives.

Web Server

The web server presents a web interface to clients and captures their MAC and IP addresses via the index.php file[8].

6.5.3 Communication Channel

The SDN controller communicates with the *Trusted Agent* using ICMP packets. Doing so limits the *Trusted Agent*'s responses to what can be contained in the ICMP packet's data field, which allows for the insertion of an additional 64 bits (or eight characters) of data.

[8]The PHP script for obtaining client MAC addresses requires that clients reside on the same subnet as the web server

This choice makes our solution easily adaptable to other controllers. Hence, messages from the *Trusted Agent* are *action*, *keyID* strings.

The *action* value is a single letter abbreviation as shown in Table 6.3 and identifies the message type (i.e., initialize, acknowledge, update, or delete) sent by the *Trusted Agent*. For messages that do not require a *keyID*, the action value is simply followed by a zero (e.g., "i,0"). However, messages from the controller may have additional data. For instance, rule insertion methods destined for the *Trusted Agent*'s Client Table will include MAC, *srcport*, *passkey*, violation, and *keyID* values. This format is recognized by the *Trusted Agent* and handled appropriately.

6.6 Test Environment and Results

Our work takes advantage of a new Mininet-based [24] platform called Mininet-WiFi [25], providing a first use case for wireless security application testing on the platform. Mininet-WiFi capitalizes on Mininet's containerized Linux kernel and adds both stations (i.e. wireless clients) and wireless APs by using *cfg80211* to communicate with wireless device drivers. It also uses the Linux 802.11 configuration API to provide communication between stations and *mac80211*.

The test environment for this implementation is shown in Fig. 6.6. With Mininet-WiFi, the wireless clients are referred to as stations while wired clients are referred to as hosts. In our test bed, there are six hosts and six stations. All six hosts are directly connected to

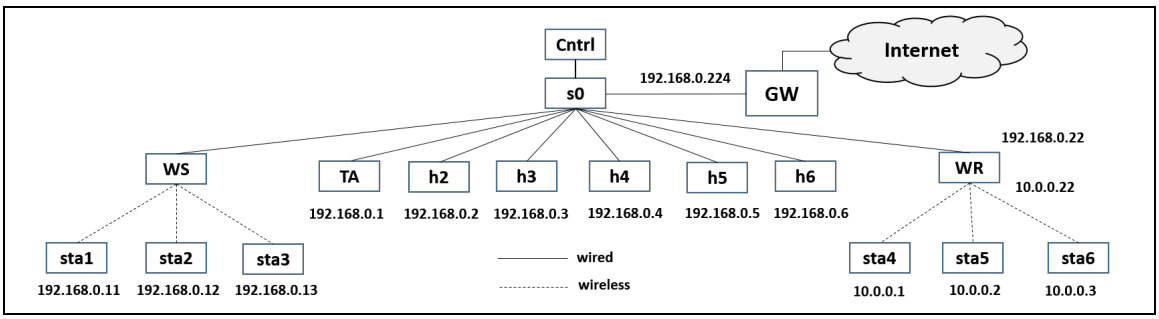


Figure 6.6: Mininet-WiFi Implementation and Testbed

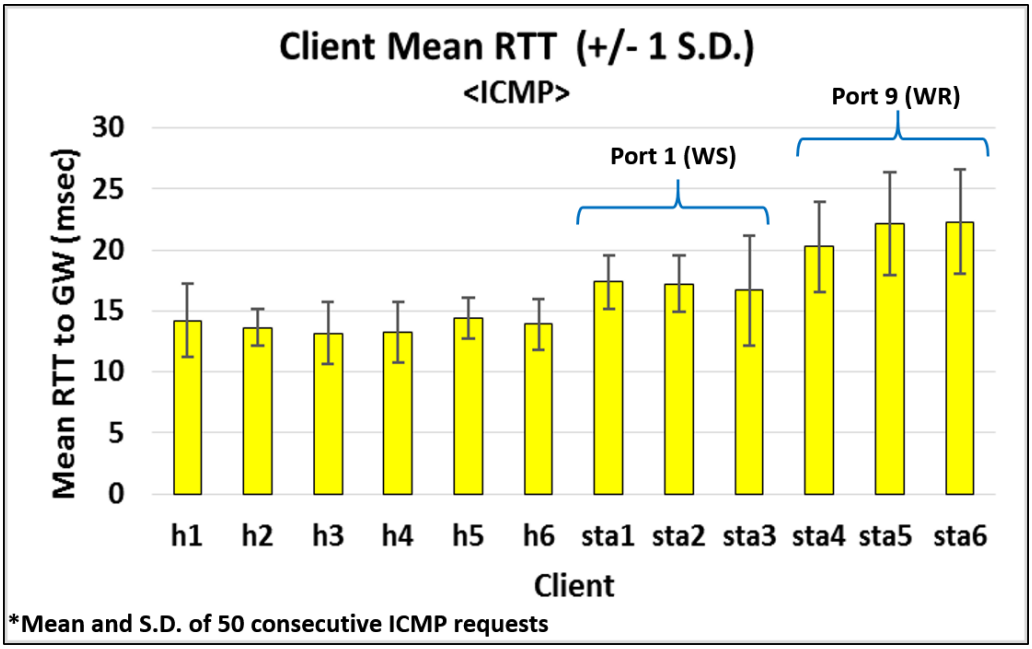


Figure 6.7: RTT from clients to gateway router

the switch, while three stations are connected to a wireless router (WR), and three other stations are connected to a wireless switch (WS). One host doubles as our *Trusted Agent*. Our work also utilizes the Ryuretic programming framework [91, 5] in conjunction with a Ryu controller and an OpenFlow Switch as part of Network Flow Guard's security implementation. The *Trusted Agent* (TA) comprises the other part of Network Flow Guard's security implementation.

Before testing Network Flow Guards security applications, we first confirm that RTTs for clients in our testbed align with results found in earlier work [132]. A total of 50 ICMP echo requests are sent from each client. The recorded results are shown in Fig. 6.7. Results for our testbed show that the wireless switches (WS) and wireless routers (WR) do, on average, impose an additional propagation cost that is higher than that of wired clients.

Network Flow Guard also requires seed and weight values to maintain its rolling port average for each port, as previously shown in Eqn. 6.1 and Eqn. 6.2. Using *iperf*, we generate TCP traffic between each client and the gateway router (GW). The rolling port average is recorded for each TCP connection. As shown in Fig. 6.8, wired clients maintain

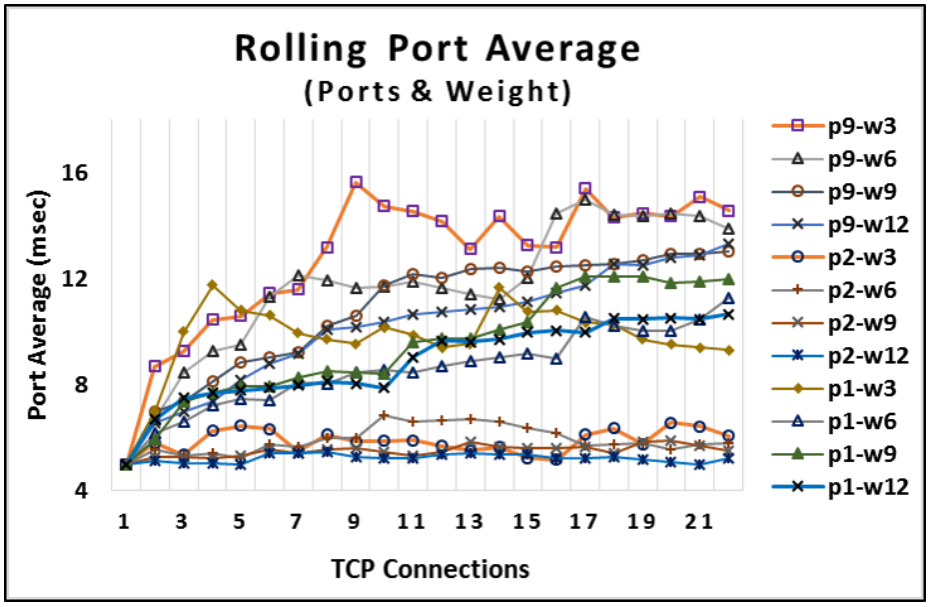


Figure 6.8: Rolling port averages at varying weights and seed of 5 msec

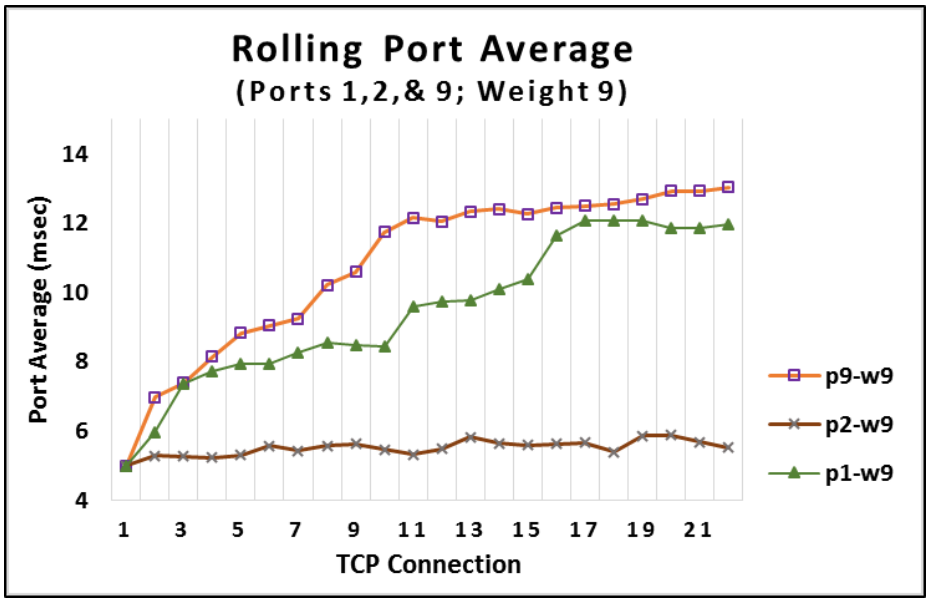


Figure 6.9: Rolling port average with weight of 9 and seed of 5 msec

a port average (or average time-to-ack) of roughly 5ms, which we choose as our seed value for Eqn. 6.1. Weight values 3, 6, 9, and 12 are evaluated on ports 1 (WS), 2 (wired host), and 9 (WR). The results for all weights are shown in Fig. 6.8, while results for our chosen weight are shown in Fig. 6.9 across the same three ports. These ports connect to the wireless switch (port 1), a wired client (port 2), and the wireless router (port 9).

Accordingly, we note that the rolling port averages for these ports quickly diverge, allowing Network Flow Guard to identify and flag the attached RAPs.

To test our Network Flow Guard security implementation, the following procedures are followed. First, the Ryuretic controller is started. Next, the Mininet-WiFi network topology is built. Then, using *xterm* [135], we open a terminal emulator for the *Trusted Agent*. Afterwards, custom scripts are run to start the *Trusted Agent's* DNS and web servers along with its ICMP communication modules.

We first test our TTL decrement module by opening a terminal emulator in station 4 (sta4), which is connected to the wireless router (WR). We then attempt to perform a curl request for a known web page. The controller immediately detects the TTL decrement caused by the WR. As a result, Network Flow Guard sets the flag field in the Policy Table to 'deny' and installs a new *Match-Action* rule to the switch to redirect traffic to its *Trusted Agent*.

The multi-MAC/IP module is tested by opening a terminal emulator for both station 1 (sta1) and station 2 (sta2). A *curl* command is executed in sta1, which completes successfully. Then, we execute a *curl* command in sta2. The appearance of a second MAC or IP (in this case, both) triggers a multi-MAC/IP violation, and Network Flow Guard sets the flag field in the Policy Table to 'deny' and installs a new *Match-Action* rule to the switch to redirect the client's traffic to the *Trusted Agent*.

In order to test the time-to-ack analysis module, we first disable the TTL decrement and multi-MAC/IP modules. Otherwise, either module will flag the RAPs before the test completes. Two methods are used to test Network Flow Guard's time-to-ack analysis module. First, *iperf*, a general command-line utility for testing the speed of TCP connections, is used to evaluate Network Flow Guard's time-to-ack analysis module. For each client (host and station), the *iperf* utility is run twelve times. The destination address for these devices is the network's gateway router (connected to port 8). As shown in Fig. 6.10, the ports connected to the wireless router (port 9) and the wireless switch (port 1) yield higher

time-to-ack values than the ports connected to wired clients. Repeated experiments yield comparable results.

Next, to better tax the system, a script is simultaneously run on all clients (host and stations) in the network, causing each client to make repeated *curl* request to one of two common web sites every three seconds. In total, each client makes 24 *curl* requests, and their port average (*portAv*) values are evaluated. The results are shown in Fig. 6.11. As with

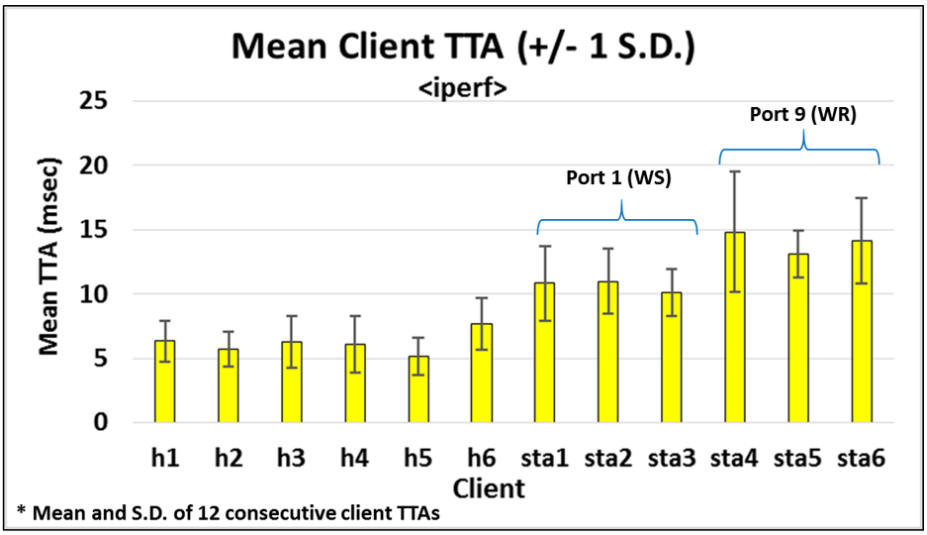


Figure 6.10: TTA analysis of iperf results per port

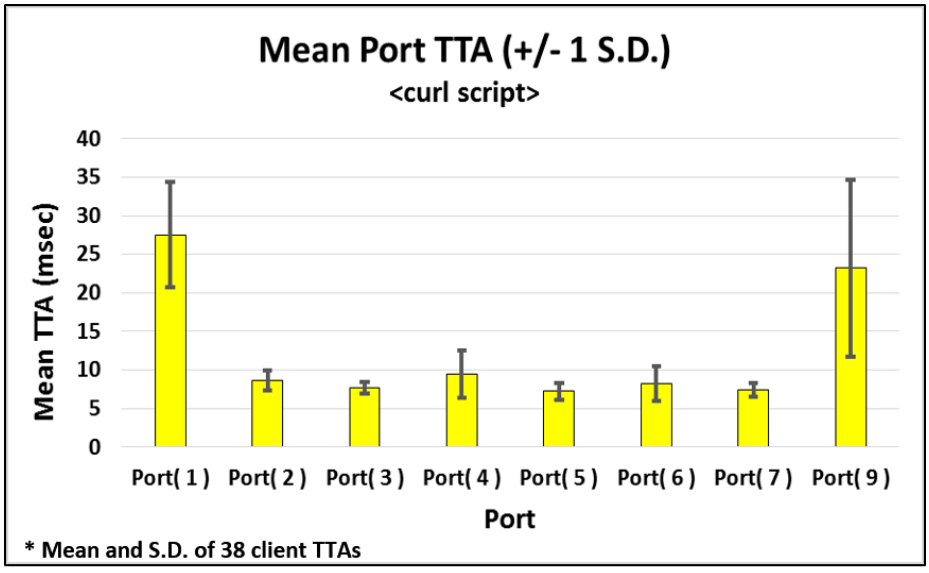


Figure 6.11: TTA analysis of curl script results per port

the *iperf* test, port averages for both the wireless switch and the wireless router are greater than those of the connected hosts. Additionally, the difference is much more pronounced as wireless clients must compete for access to a single port. The rolling port average is also shown for each port in Fig. 6.12. While port 1 (connected to the WS) shows an immediate deviation from other ports, port 9 (connected to the wireless router) takes several more iterations to become comparably distinguishable. Accordingly, Network Flow Guard first flags the wireless switch and then the wireless router for active detection.

To test our Active Detection Engine (ADE), we intentionally flag the port connected to our wireless router for 'test' within the Policy Table. This too sets a *Match-Action* rule to the switch that temporarily redirects traffic from port 9 to Network Flow Guard's *Trusted Agent*. A terminal emulator is opened for station 4 (sta4) and Firefox is used to open a web page. Monitoring traffic for both the wireless router and the *Trusted Agent* with Wireshark reveals that sta4's browser submits a DNS request that is intercepted and passed to the *Trusted Agent*. The *Trusted Agent* provides a DNS response. The browser then establishes a TCP connection with Network Flow Guard's *Trusted Agent*, and an index.php file is delivered to the browser. For the purpose of this experiment, the index.php file is

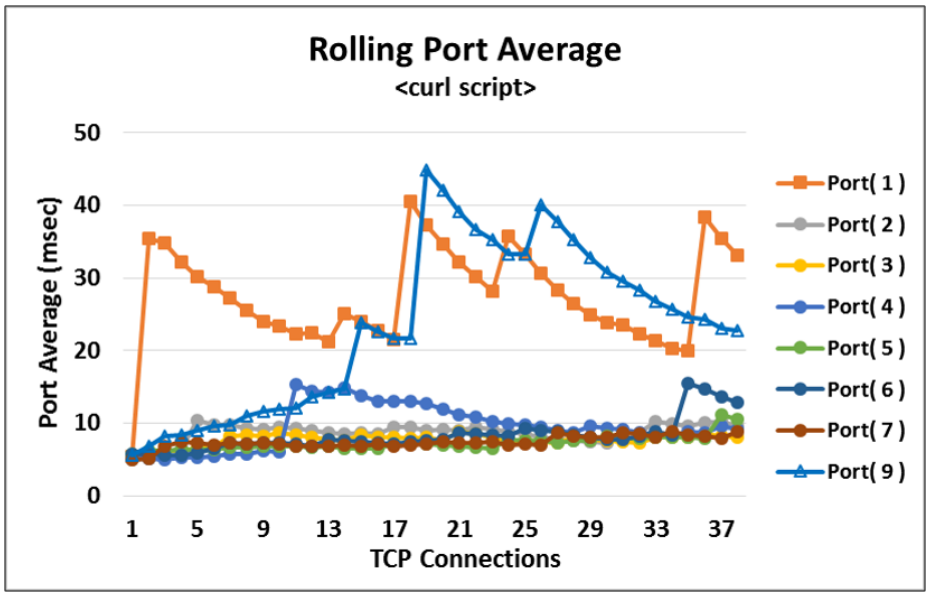


Figure 6.12: Rolling average analysis of curl script results per port

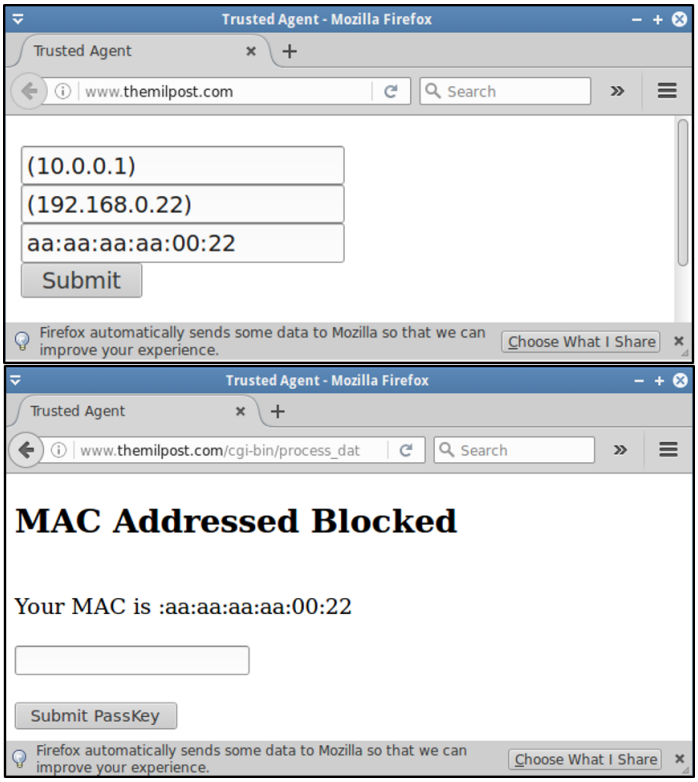


Figure 6.13: WebRTC Results (Firefox Web Browser)

modified to display the client's local IP address, as seen by the *Trusted Agent* and the client's browser along with a submit button. Normally, this is all hidden from the client and the form submission is automatic. As shown in Fig. 6.13 for sta4, two different IP addresses are displayed. Once submitted, the Common Gateway Interface script identifies the mismatch and sets the *keyID* in the RAP test file to 'fail'. The *Trusted Agent* then sends the results to the SDN controller, and Network Flow Guard sets the flag field in the Policy Table to 'deny'. Future attempts by sta4 to access network services yields the *Trusted Agent*'s *passkey* page. Once the client addresses the issues causing its flag, a *passkey* is provided via help desk. Entering the *passkey* allows the client to regain their privileges.

As discussed earlier, webRTC is not yet compatible with all browsers (e.g., Internet Explorer, Safari, etc.). When Network Flow Guard attempts an active test on these browsers, it receives a pre-set value: 'test'. This notifies Network Flow Guard that an incompatible browser was used. No action is taken against the client.

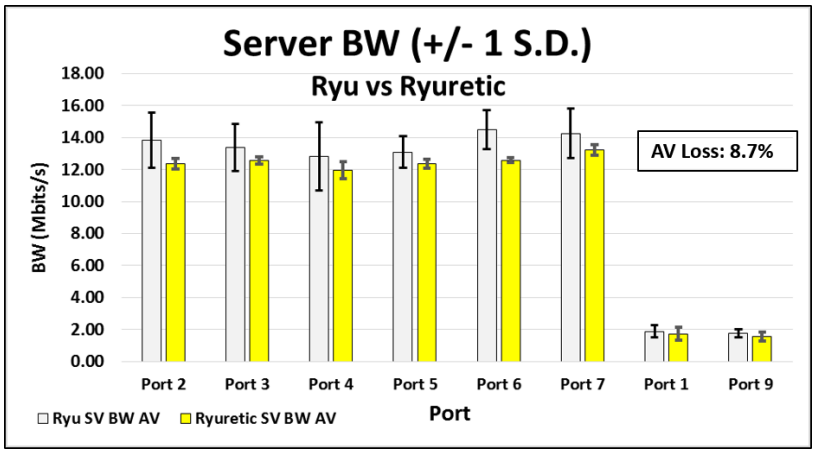


Figure 6.14: Server BW comparison of Ryu vs Ryuretic

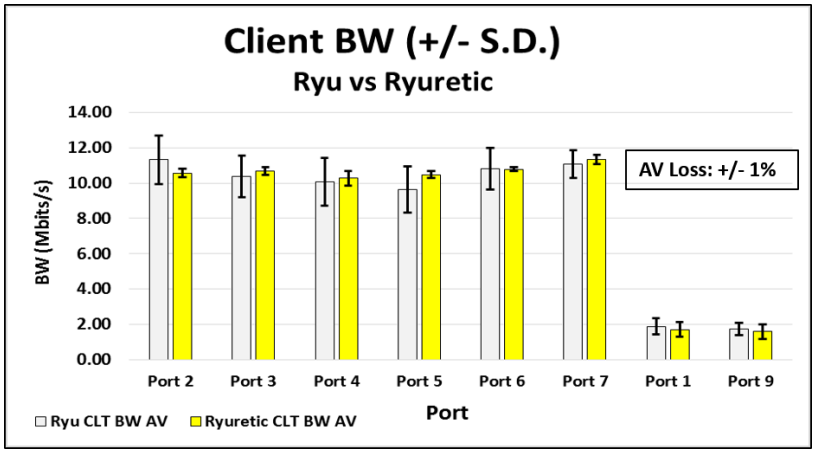


Figure 6.15: Client BW comparison of Ryu vs Ryuretic

We evaluate Network Flow Guard's performance cost against a Ryu controller, implementing a simple, MAC-learning, switch, which permanently installs *Match-Action* rules to the switch as each device connects to the network. Hence, once rules are installed, the switch handles all forwarding. However, SDN security applications must process extra packets at a cost to performance. To determine this cost, we utilize *iperf* to conduct bandwidth tests for each port seven times. Fortunately, Network Flow Guard imposes minimal impact to the network's performance. As shown in Fig. 6.14, the performance cost to flows from the server is 8.7%, while client cost, as shown in Fig. 6.15, is negligible.

6.7 Discussion

Our work offers a solution for detecting RAPs that incorporates the emerging webRTC API and other traditional approaches into the dynamic capabilities of an SDN. To our knowledge, this work represents a first attempt to utilize an SDN to detect RAPs. Likewise, it uses the webRTC API in a disruptive way to detect hidden subnets. Additionally, our per flow approach to passive RAP detection reduces the total compute of our controller by evaluating just three packets of a TCP handshake (i.e., TCP[SYN], TCP[SYN,ACK], and TCP[ACK]) before installing temporary *Match-Action* rules to the switch, allowing the TCP flow to complete without additional controller involvement. Moreover, our active detection measures are able to detect NAT devices without evaluating HTTP(S) strings and raising privacy concerns. Instead, HTTP GET requests are sent to Network Flow Guard's *Trusted Agent*, which delivers our custom scripts to the client's browser via an index.php file. Once the client's browser verifies its local IP address, the SDN controller then directs future packets to their appropriate destination.

Another benefit of Network Flow Guard's use of SDN is a centralized monitoring point. One issue that Yasemin et al. [102] identify in their work is that the number of hops between the machine observing traffic and the device generating the analyzed traffic may not be known. For SDN, however, there is a centralized monitoring point since the client plugs directly into the switch responsible for capturing traffic for analysis. As a result, the TTL should arrive at the switch unaltered. Of course this does require that NATs play fair. For instance, if TTL decrement is disabled (allowed by Juniper routers, etc.), then TTL detection measures can be evaded. However, we believe that, while TTL decrementing can be circumvented, detecting it should still be part of any NAT detection scheme, and Network Flow Guard includes this capability.

While tracking the time-to-ack from clients was affective for identifying potential RAPs in our test environment, we note that our initial seed value of *5ms* may have to be adjusted

for other networks. Still, the time-to-ack analysis module is only intended to flag potential RAPs for testing, so clients are not denied access to the network because of this feature. Additionally, we are not able to duplicate the scenario where the RAP sends TCP[ACK] messages to the destination on behalf of its client. For these packets, their time-to-ack would be equivalent to other hosts on the network. Yet, as we observe from our Wireshark captures (discussed in Section 6.4.2), the packets comprising the three-way handshake of its TCP connections are handled by the client, so even this device will be flagged for 'test' by our time-to-ack analysis module. Better yet, the presence of a decremented TTL field also means that it will be flagged for 'deny' upon transmission of its first IP packet.

Still, our method does not protect clients from connecting to RAPs. Instead, it blocks RAPs at the port, which is an approach taken for NAT devices as well. Additionally, while it is possible that IPv6 could address many of the short comings that make NAT devices necessary, for the moment, it appears that backwards compatibility with existing protocols (i.e., IPv4) is preventing IPv6 from gaining significant traction in network operations. Consequently, RAPs also function within IPv6 architectures, so methods for detecting them are still needed.

Another point worth mentioning is that Ryu [32] was not our original choice for this implementation. We originally intended to use POX [31] in conjunction with Pyretic [16]. This modular programming language is very intuitive for users and greatly simplifies application development for SDNs. Unfortunately, Pyretic's dependence on POX, which is only compatible with OpenFlow version 1.0, did not allow us access to the TTL field in IPv4 headers. Hence, to further pursue our work, we turned to Ryu, which supports OpenFlow versions 1.0-1.5. However, Ryu lacks the abstractions available in Pyretic, which made modular application development much more complicated. This inspired our earlier work to create the Ryuretic framework for modular application development [5, 91]. However, an additional feature was needed in the Ryuretic framework to simplify the process of dynamically redirecting and modifying packet header fields at the switch. That feature

is added in this work, and the updated Ryuretic platform can be found at [91].

Mininet-WiFi [25] was also not our original choice for our test environment; yet, its discovery proved particularly timely for this research. Many researchers use Mininet [24] to setup their testbeds, and it is a useful tool for testing security applications for a wired network. However, it offers few features for emulating wireless devices (e.g., wireless switches, wireless routers, and stations) on SDNs. Without Mininet-WiFi, we would have been relegated to setting up our testbed in an ns-3 [136] topology or OpenNet [137]. And, while ns-3 is an excellent resource, it introduces another platform that researchers may not be familiar with, and it lacks the ability to stand-up multiple clients simultaneously or emulate various servers. OpenNet [137] was the emulation platform we originally considered. It represents a recent attempt at modeling wireless devices and serves as a software-defined wireless LAN (SDWLAN) simulator. OpenNet implements taps between Mininet nodes and wireless ns-3 APs that simulate wireless features. Unfortunately, the drawback to this platform is that no clear procedure exists for adjusting link parameters, such as packet loss, delay, and channel bandwidth without modifying OpenNet's source files. Fortunately, Mininet-WiFi does include these features and a means to manipulate interference, data loss, delay, and other features within the Mininet-WiFi topology file.

6.8 Future Work

While this research implements a viable solution for detecting and preventing RAPs on SDNs, there is still more that can be done. For instance, this framework, introduces the concept of a *Trusted Agent*, which serves as part of Network Flow Guard's Active Detection Engine (ADE) while also automating the process for clients to regain network privileges. Yet, the *Trusted Agent* could potentially perform many other yet unrecognized tasks. Additionally, detecting and denying a RAP when a single client is connected to a wireless switch remains an open challenge in our work.

6.9 Conclusion

We consider the capabilities of traditional networks to defend against rogue access points (RAPs), and then offer a first-ever, SDN-based approach to detecting and preventing RAPs. An emerging architecture, webRTC, is also exploited as a novel method for detecting subnets hidden behind NAT devices. Hence, our work contributes a new RAP detection method that is applicable to both SDNs and traditional networks. Other contributions of our work includes its use case for the newly developed wireless network emulation framework (i.e., Mininet-WiFi) and its introduction of a new feature enabling Ryuretic (a modular programming language for SDN application development) to simultaneously redirect packets and modify their header fields. Using Mininet-WiFi, we implement a testbed capable of supporting wireless switches, wireless routers, and wireless hosts (stations). In doing so, our work further validates Mininet-WiFi's viability towards wireless security application development and testing. Additionally, our work adds a new operation feature (i.e., modify) to the Ryuretic programming language to enable *port intercepting* as a means to perform active testing on suspected clients. All of these contributions work together to demonstrate a hybrid testing methodology consisting of passive and active techniques for detecting RAPs connected to an organization's network with a performance cost of less than 9%. All files involved in our work and setup instructions are available on GitHub [91].

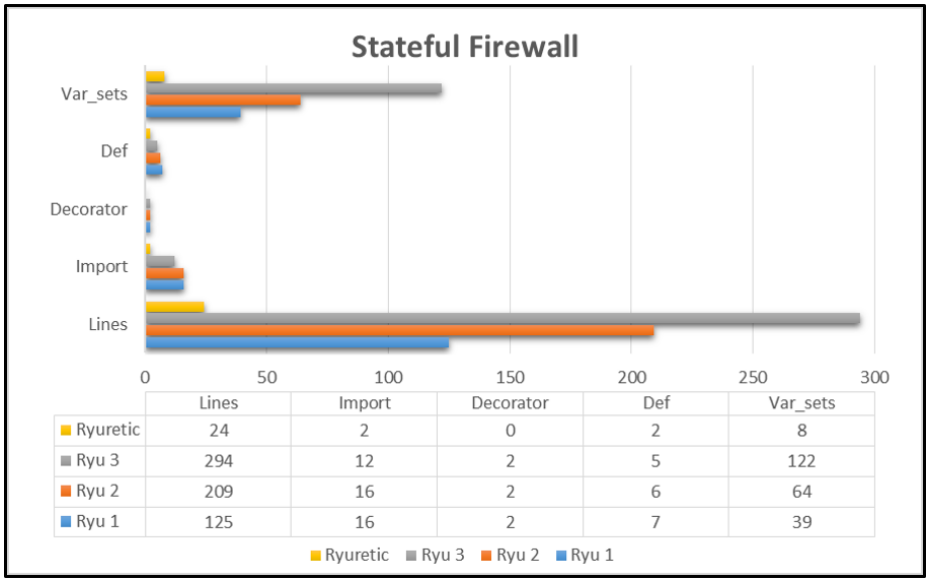
CHAPTER 7

CONCLUSION

7.1 Other Results

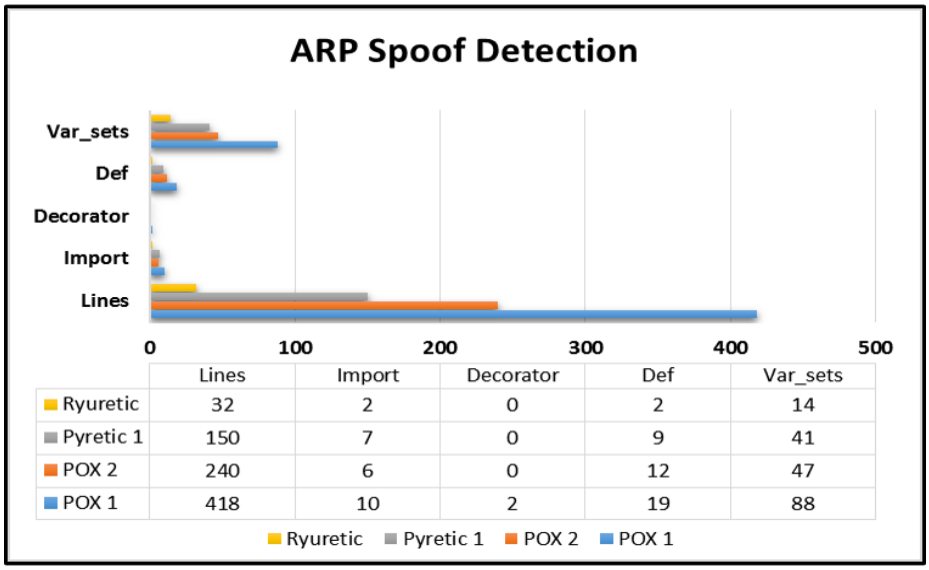
Ryuretic offers comparable performance overhead to other programming frameworks. For instance, in Chap. 6, we observe that the performance overhead of Network Flow Guard while monitoring TCP traffic for rogue access point indicators (i.e., observing time-to-ack values) on an Open vSwitch is roughly 9%. This value is comparable to the performance of another SDN-based framework known as FRESCO [138, 139]. Using a Nexus 5 and a Nexus 7 for their FRESCO security monitoring applications, Hong et. al reported performance overheads of 9% and 7% respectively.

Ryuretic also offers programmers significant abstractions and less coding to achieve comparable results to Ryu-based programs. For instance, when building a stateful firewall, as seen in Fig. 7.1, Ryuretic requires far less lines of code (80% less than its closest rival)



	Lines	Import	Decorator	Def	Var_sets
Ryuretic	24	2	0	2	8
Ryu 3	294	12	2	5	122
Ryu 2	209	16	2	6	64
Ryu 1	125	16	2	7	39

Figure 7.1: Code comparison of Ryuretic and Ryu based Stateful Firewalls



	Lines	Import	Decorator	Def	Var_sets
Ryuretic	32	2	0	2	14
Pyretic 1	150	7	0	9	41
POX 2	240	6	0	12	47
POX 1	418	10	2	19	88

Figure 7.2: Code comparison of Ryuretic and POX and based Stateful Firewalls

to implement its stateful firewall. We also observe that Ryuretic requires no decorator statements and only 13%-16% of the import statements. Likewise, Ryuretic requires less methods to be created. Moreover, the variables that must be declared and set in Ryuretic are far fewer (80%-94%) than those in the Ryu programs. This is because the Ryuretic coupler or backend already handles the decorators, imports, variables, and methods needed to create an application atop the Ryu controller.

Similar observations are also seen when Ryuretic is compared with POX and Pyretic programs (see Fig. 7.2). While decorators can be abstracted from POX and Pyretic programs, these programs still contain significantly more lines of code, imports, unique methods, and set variables compared to a Ryuretic program. For instance, Pyretic requires nearly five times more lines of code to implement a comparable ARP spoof detection method, while POX requires eight times more lines than Ryuretic. Similarly, Ryuretic uses three to five times less imports and four to ten times less definitions. Ryuretic also uses significantly less variables, ranging from a factor of three to seven.

In RYU and the other programming examples, the programmer is responsible for every line of code; however, Ryuretic provides a framework that allow programmers to simply

focus on inserting lines of code for interacting with a packet (*pkt*) object and returning match:action decisions in the form of *fields* and *ops* objects. This object based programming framework allows network operators to focus on the specific task they hope to achieve rather than the inner-workings of an underlying controller.

7.2 Discussion

Throughout this dissertation, we highlight how software-defined networking (SDN) [10] allows for a single controller to orchestrate the actions of an entire network of switches[1]. Meanwhile, southbound interfaces, such as OpenFlow [12], provide network operators with a single, vendor-agnostic interface for creating network applications, allowing for more fine-grained orchestration. In addition, these interfaces are further augmented by programming frameworks, like Pyretic [16] and Ryuretic [5] to provide greater abstraction and shield network operators from the complexities inherent in network application development. With these abstractions, network operators are able to focus more on the applications they want to implement, and less on the inner workings of the network. Furthermore, as organizations seek to protect their clients, data, and network resources, researchers and network operators are looking towards SDN's capabilities to quickly produce network security applications that address various attack vectors as they are discovered.

With regard to network security, many researchers [140] already believe that SDN may very well prove to be one of the most impactful technologies for driving a variety of innovations. Consequently, SDN has already been leveraged to thwart DDoS attacks [141, 129, 142], provide dynamic access control [143], add security monitoring as a service [144], provide security for BYOD initiatives [139], and check for potential violations of key network invariants [145]. Likewise, industry is deploying SDN to strengthen network architectures and improve performance and utilization [146], reduce operational cost [147], and enable the rapid creation of new network applications and functions [148]. Other bene-

[1]This dissertation only considers a single controller, though distributed, logically centralized controllers can be used for more robust control options (e.g., fault tolerance, scalability, and etc.).

fits of this paradigm include greater vendor neutrality, reduced middlebox dependence, and less network operator involvement with day-to-day configurations.

Accordingly, this work takes advantage of observations by Shin et al. [140] that indicate how SDN can clearly enhance network security functions in specific ways. First, the ability to dynamically control network flows allows greater flexibility in security application deployments without the need for additional middleboxes. Second, its centralized view of the network creates great opportunities for network-wide security monitoring. Likewise, this ability allows SDN security functions to more quickly and efficiently comprehend network attacks (even those that are widely distributed in the Internet) than do legacy network security monitoring systems [140]. Third, SDN can be leveraged for the rapid prototyping of new and advanced network security features without adding extreme cost or effort.

This ability to rapidly prototype solutions serves as a key enabler for our own research. In doing so, we systematically develop a platform for simplifying application development, automating the updates of policy enforcements in accordance with specified security policies, and allowing for the incorporation of active testing measures within an SDN-based, security/management framework. Initially, the solutions implemented in this dissertation focus on detecting and denying violations of network policies. Many other security solutions, as revealed in our research, adapt this approach as well. However, such solutions merely detect and block clients who violate specified policies. They give little consideration to the additional work that these solutions impose on network operators who must manually reconfigure policy enforcements once they are triggered.

To avoid network operator involvement, some researchers have developed solutions that force the SDN controller to supplant network protocols, such as DHCP and ARP. In one case [41], the SDN controller handles all DHCP requests within a virtual environment in order to deny rogue DHCP servers access to the DHCP-requests of other clients. Similarly, Kang et al. [71] leverage an SDN controller to capture all MAC addresses as virtual devices are created in a cloud and then respond to ARP requests itself in order to cut down on

ARP traffic in the network. In other cases, researchers simply stop at the detection of a security violation–leaving it to the network operator to investigate further and implement configurations as required. In some other cases, researchers allow policy enforcements to timeout without any verification that the cause of the original violation was addressed [138, 140].

This dissertation began with similar efforts, which allowed us to demonstrate that it is possible to implement security solutions that do not require the controller to monitor for and respond to every packet. Instead, our solutions enhance existing protocols by monitoring packets and acting when a policy violation occurs. Hence, our earlier solutions from Chap. 2 and Chap. 3 demonstrate that it is relatively simple to incorporate SDN security solutions that facilitate both static and dynamic configurations simultaneously. Moreover, both of these solutions are applicable to both physical and virtual network infrastructures.

However, by not incorporating a means to dynamically update policy enforcements in an automated manner, these solutions also create additional work for network operators. Not only must they investigate the reason for a client being denied access to network services, they must also perform manual configurations to reinstate the client once the client gains approval to rejoin the network. Or, they must investigate a flagged client and manually transition them to a state of 'deny' or 'allow' based on further investigation. But, doing so increases the possibility of a network operator causing additional network errors, a temporary loss of orchestration, and/or a partial or permanent loss of network state. None of these outcomes are desirable. Upon completion of these security applications, we also reached a limit of what we could accomplish with Pyretic [16] and the OpenFlow 1.0 [12] southbound interface.

The security features we next sought to implement required access to additional packet header fields (i.e., time-to-live) and a means to support active testing and policy enforcement transitions. Unfortunately, these are requirements that Pyretic and POX simply cannot support. What's more, these desired security features also require communication chan-

nels, which are also unavailable with Pyretic and POX. These limitations led us to develop the Ryuretic [5] programming framework for Ryu. Ryuretic allows network operators to provide object-based instructions that allow their applications to match on specific packet header fields (having over 40 options) and then assign a variety of actions to individual flows, which include forward, drop, redirect, mirror, and modify (simultaneously changes packet header fields and redirects network flows). Consequently, beyond accessing additional packet header fields, the Ryuretic programming language also offers programmers the ability to craft packets. Furthermore, Ryuretic also allows network operators to setup proactive and reactive policies that target specific layers of the OSI model within their networks. These features make Ryuretic a key enabler for our transition framework introduced in Chap. 5 and for our active testing features introduced in Chap. 6.

It is in Chap. 5 of this dissertation that we finally address the need for an automated security policy transition framework for network operators. Chap. 5 also introduces a Trusted Agent and a limited ICMP-based communication channel. The Trusted Agent essentially serves to validate that a client has met network policy requirements before sending a policy enforcement revocation request to the SDN controller to reinstate the client's privileges. As we discussed in Chap. 5, this work frees network operators from daily reconfiguration requirements, which are caused by client policy violations, by allowing the Trusted Agent, or a less-skilled help desk attendee, to work with the client and address the violation in order to obtain a passkey.

In Chap. 6, we further enhance the transition framework and Ryuretic to support policy enforcement state changes. Those being 'test', 'allow', and 'deny'. Unlike other research where the network operator is called upon to make decisions about flagged clients, this chapter utilizes the Trusted Agent to transition a client from a state of 'test' to a state of 'allow' or 'deny'. In Chap. 6, a solution that uses a port interception technique made possible by Ryuretic's modify (mod) operation is used to redirect the client's HTTP requests to the Trusted Agent. The Trusted Agent, in turn, delivers a PHP/JavaScript file to the

client's browser, which exploits the webRTC architecture to determine whether the client is residing on a hidden subnet.

Additionally, within the context of this work, clients are considered to be untrustworthy. As a result, none of the solutions implemented in this dissertation require client systems to participate beyond interacting with our Trusted Agent to regain network privileges once they are lost. Hence, network operators do not need to worry with maintaining client-side software to utilize any of the security solutions presented in this work–the one exception being that our webRTC solution requires a Chrome, Opera, or Firefox browser be used when detecting hidden subnets. However, this is software that many clients (over 1 billion) already use on their own systems, and we expect more browsers to soon enable the webRTC architecture as well.

Finally, while we believe that this work offers significant contributions to network security, potentially reduces the number of middleboxes required on a network, and eliminates a significant number of network operator configuration requirements, the Network Flow Guard framework is not an end all solution. In truth, its security solutions are only intended as an initial barrier or a first defense in a defense-in-depth strategy. As such, it has little application for security within the higher echelons of network infrastructure. At least, those applications are not considered in this work. Moreover, network operators must still carefully evaluate their policies to avoid policy conflicts.

7.3 Contributions

This work originally set out to determine whether it is possible and then how to leverage the capabilities of SDN, aided by NFV, to enhance network security. In doing so, we first demonstrate that SDN security features can indeed be rapidly developed to address traditional security challenges in both static and dynamic network environments. This work also offers a programming framework allowing network operators to match on over 40 packet header fields and then choose such options as forward, drop, redirect, mirror, modify, or

craft packets. Besides enabling fine-grained and targeted application development, this framework makes possible an ICMP-based communication channel and a port interception method. Doing so allows this work to introduce a Trusted Agent and alleviate network operator troubleshooting and network configuration requirements by handling client validations and automating the revocation or updating of policy enforcements with the SDN controller. Finally, this work augments the Trusted Agent to not only revoke policy enforcements, but to guide state transitions from 'test' to 'allow' or 'deny'. This allows for the creation of a security system deploying both passive and active testing methods and capable of both automatically removing client access from network services and reconnecting those services when the client addresses the violation. Additionally, a new Mininet-based emulation environment, Mininet-WiFi, is utilized and validated as a legitimate platform for testing wireless security applications. Furthermore, this work introduces a novel feature for NAT detection using the webRTC architecture to detect hidden subnets.

7.4 Future Work

In concluding this dissertation, we summarize some potential and interesting research directions that this work could support. Possible directions include enhancing the Trusted Agent with machine learning algorithms, incorporating an east-west bound interface in the Ryuretic framework to enable cross-domain policy enforcement or verification, detecting and resolving network policy conflicts, and expanding the Trusted Agent to take on more responsibilities and implement additional security features.

For instance, machine learning approaches to handling client interactions by the Trusted Agent would allow our Trusted Agent to behave in a much more robust manner. Goals of such research would be to enhance the user interface presented by the Trusted Agent to provide a more human interaction and better decision making capabilities. Likewise, machine learning could enhance the Trusted Agent to better coordinate policy agreements and validate them across multiple domains.

The incorporation of an east-west bound interface would also serve to support multi-domain policy enforcement amongst multiple SDN controllers. Likewise, it would allow for more robust communication between Network Flow Guard's Trusted Agent and the SDN controller. Currently, the communication channel offered by Network Flow Guard's transition system is 64 bytes, allowing for eight characters. With a better in-band communication protocol, Trusted Agent to SDN controller communication and SDN controller to SDN controller communication can be much more robust, possibly supporting encryption. Already, we are looking at modifying Ryuretic to work with OpenFlow switches that support the REST API. However, the inclusion of the REST API is only in its nascent stages. It is possible that the OpenFlow protocol could be further extended to allow for SDN communication messages for the controller as well.

As mentioned previously, the Network Flow Guard platform does not yet include a policy conflict detection/prevention mechanism. As a result, multiple applications could potentially circumvent the policies introduced by other applications. Currently, it is left to the network operator to choose a priority for these rules, but in cases where priorities are equal, there is no resolution protocol for handling such issues. Hence, Network Flow Guard could benefit from services that check for potential violations using similar methods as Veriflow [145].

Finally, with the inclusion of a Trusted Agent and the ability to perform active testing, future work can also include more robust network security and management measures. The Trusted Agent could potentially coordinate its efforts with other middleboxes on the network or include its own measures for augmenting the passive detection algorithms utilized by the SDN controller. It is also possible that the Trusted Agent could take a more active role in validating network paths before allowing the SDN controller to modify traffic flows.

7.5 Conclusion

In this dissertation, we introduce the capabilities of SDN and NFV, and systematically investigate their application towards security. As observed throughout this dissertation, a wide swath of research in SDN-based security applications are steadily being introduced. However, despite SDN's ability to address numerous security vulnerabilities, developing and deploying complex SDN-based security services remains a significant challenge–particularly for network operators who must maintain these systems and their policy configurations.

Hence, we demonstrate how SDN with NFV can better empower network operators to better defend and manage their networks in a consistently evolving attack landscape with better tools and abstractions. Towards this end, we present the Network Flow Guard architecture as the culminating system of this work, which includes the Ryuretic programming framework and the security policy transition framework. We also present several use cases where SDN, working with NFV, is quickly adaptable to detect and deny security policy violations (i.e. rogue DHCP servers, ARP poisoning, and NAT/RAP devices). Furthermore, we address a key problem for network operators involving the continual and manual manipulation of network policy configurations.

We also believe that Ryuretic serves as powerful programming framework offering key abstractions for rapidly prototyping and delivering innovative security and management applications for the growing number of SDNs. Likewise, our own evaluations indicate that Ryuretic offers similar overhead costs as other comparable programming frameworks. However, Ryuretic does offer unique features that allow for dynamic and automated policy enforcement transitions (i.e., the craft packet feature). Accordingly, this work serves to push SDN and NFV towards becoming one of the most impactful technology combinations for reducing network operator workloads while driving innovation in network security.

Appendices

APPENDIX A
EXPERIMENTAL EQUIPMENT

As mentioned earlier, virtualized network functions enable much of the work completed in this dissertation. We discuss the tools utilized below.

A.0.1 Hardware

Lenovo (W540) ThinkPad

Having 16 GB RAM, 2.40GHz Intel(R) Core(TM) i7-4700MQ CPU (8 CPUs), this computer serves as the primary hardware for this work's emulations.

TP-Link, 8-Port Gigabit Easy Smart Switch, Model TL-SG108E

This switch offers a tap port that we use to monitor network flows from various wireless routers and wireless switches to deduce which devices decremented the time-to-live in packet headers or exposed multiple MAC and IP addresses.

Wireless Routers and Wireless Switches

See Table 6.2.

A.0.2 Virtual Environment

Oracle VM Virtual Box

Used to run the virtual machines created for developing and testing the network security and management solutions developed in this work.

Virtual Machines

The virtual machines created in this work all use as their base the All-in-one SDN App Development Starter VM, which is downloadable from www.sdnhub.org. It uses a Ubuntu 14.04 operating system, and comes preinstalled with a number or SDN controllers and emulation environments.

Emulators

In this work, research was primary conducted in one of two virtual environments. Those are Mininet [24] and Mininet-WiFi [25].

A.0.3 Tools

Lighttpd Web Server

Used with the Trusted Agent to render PHP/JavaScript encoded files to clients for both client requirement validation and active testing.

DHCP Server

ISC-DHCP-server. Serves as our primary DHCP server. It is an internal Ubuntu network service that enables host computers to be automatically assigned settings from a server as opposed to manually configuring each network host.

UDHCPD. Serves as our rogue DHCP server. It is a very small DHCP server that receives DHCPDISCOVER packets and responds with DHCPOFFER packets.

Scapy.py

Scapy is a powerful interactive packet manipulation program. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. We use this python-based program to create many of the servers required for our Trusted Agent as well as to generate packets for testing.

REFERENCES

[1] H. Kim, T. Benson, A. Akella, and N. Feamster, "The Evolution of Network Configuration: A Tale of Two Campuses," ACM, Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference, 2011, pp. 499–514.

[2] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark, "Kinetic: Verifiable dynamic network control," Oakland, CA: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), May 2015, pp. 59–72, ISBN: 978-1-931971-218.

[3] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management.," NSDI, 2009, pp. 335–348.

[4] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "Pisces: A programmable, protocol-independent software switch," *AT&T Research Academic Summit, Bedminster, NJ, USA*, 2016.

[5] J. Cox, S. Donovan, R. Clark, and H. Owen, "Ryuretic: A modular programming language for ryu," IEEE, Military Communications Conference, 2016. MILCOM 2016. IEEE, 2016.

[6] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: An intellectual history of programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[7] A. Aguado, V. Lopez, J. Marhuenda, Ó. G. de Dios, and J. P. Fernández-Palacios, "Abno: A feasible sdn approach for multi-vendor ip and optical networks," Optical Fiber Communication Conf.(OFC), 2014.

[8] S. Perrin and S. Hubbard, *Practical Implementation of SDN & NFV in the WAN*, [Online]. Available: `http://events.windriver.com/wrcd01/wrcm/2016/08/WP-practical-implementation-of-sdn-nfv-in-the-wan.pdf`, 2013.

[9] O. N. Fundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, 2012.

[10] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, vol. 17, no. 2, pp. 30–32, 2009.

[11] S. Hong, L. Xu, H. Wang, and G. Gu, "Poisoning network visibility in software-defined networks: New attacks and countermeasures," NDSS, 2015.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[13] K Giotis, C. Argyropoulos, G. Androulidakis, D. Kalogeras, and V. Maglaris, "Combining openflow and sflow for an effective and scalable anomaly detection and mitigation mechanism on sdn environments," *Computer Networks*, vol. 62, pp. 122–136, 2014.

[14] S. Lim, J.-I. Ha, H. Kim, Y. Kim, and S. Yang, "A sdn-oriented ddos blocking scheme for botnet-based attacks," IEEE, Ubiquitous and Future Networks (ICUFN), 2014 Sixth International Conf on, 2014, pp. 63–68.

[15] I. O. De Urbina Cazenave, E. Köşlük, and M. C. Ganiz, "An anomaly detection framework for bgp," IEEE, Innovations in Intelligent Systems and Applications (INISTA), 2011 International Symposium on, 2011, pp. 107–111.

[16] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular sdn programming with pyretic," *Communications Magazine, IEEE*, vol. 38, no. 5, pp. 128–134, 2013.

[17] *RYU*, [Online]. Available: `http://osrg.github.io/ryu/`.

[18] K. Marco, *InformationWeek 2014 State of the Data Center Survey*, [Online]. Available `http://reports.informationweek.com/abstract/6/12525/Data-Center/research-2014-state-of-the-data-center.html`, 2014.

[19] PiperJaffray, *2015 Piper Jaffray CIO Survey*, [Online]. Available: `https://piper2.bluematrix.com/sellside/EmailDocViewer?encrypt=7856c68e-3f1a-4ce9-a7e7-99fe25145cd9&mime=pdf`, 2015.

[20] S. Donovan and N. Feamster, "Intentional network monitoring: Finding the needle without capturing the haystack," ACM, Proceedings of the 13th ACM Workshop on Hot Topics in Networks, 2014, p. 5.

[21] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," ACM, vol. 43(4), ACM SIGCOMM Computer Communication Review, 2013, pp. 27–38.

[22] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service," *SIGCOMM Comput. Commun. Rev.*, vol. 42, no. 4, pp. 13–24, Aug. 2012.

[23] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[24] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," ACM, Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, 2010, p. 19.

[25] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-wifi: Emulating software-defined wireless networks," IEEE, Network and Service Management (CNSM), 2015 11th International Conference on, 2015, pp. 384–389.

[26] V. Sekar, "Enabling software-defined network security for next-generation networks," in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, ACM, 2016, pp. 1–1.

[27] D. Kushner, "The Real Story of Stuxnet," *IEEE Spectrum*, vol. 50, no. 3, pp. 48–53, 2013.

[28] D. Kreutz, F. Ramos, and P. Verissimo, "Towards Secure and Dependable Software-Defined Networks," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ACM, 2013, pp. 55–60.

[29] E. Auchard, *Cisco Router Break-Ins Bypass Cyber Defenses*, [Online]. Available: `http://www.reuters.com/article/2015/09/15/us-cybersecurity-routers-cisco-systems-idUSKCN0RF0N420150915?feedType=RSS&feedName=businessNews`, 2015.

[30] J. Cox, R. Clark, and H. Owen, "Security Policy Transition Framework for Software Defined Networks," IEEE, The First International Workshop on Security in NFV-SDN, 2016. SNS 2016. IEEE, 2016.

[31] *Pox*, [Online]. Available: `http://www.noxrepo.org/pox/about-pox/`.

[32] S. Ryu, "Framework community,," *Component-based software defined networking framework*, 2014.

[33] J. H. Cox Jr, R. J. Clark, and H. L. Owen III, "Leveraging sdn to improve the security of dhcp," ACM, Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, 2016, pp. 35–38.

[34] J. H. Cox, R. J. Clark, and H. L. Owen, "Leveraging sdn for arp security," IEEE, SoutheastCon 2016, 2016, pp. 1–8.

[35] D. O'Connor, *Dhcp snooping–filter those broadcasts!* [Online]. Available: https://mellowd.co.uk/ccie/?p=5796, 2014.

[36] T. O'Connor, *Personal communication*.

[37] T. Oconnor, *How to find a rogue dhcp server on your network*, [Online]. Available: http://tomoconnor.eu/blogish/how-to-find-rogue-dhcp-server-your-network/, 2013.

[38] C. Meraki, *Tracking down rogue dhcp servers*, 2015.

[39] WS, *Wireshark*, [Online]. Available: https://www.wireshark.org/.

[40] Shrubbery, *Rancid - really awesome new cisco config differ*, Shrubbery Networks, INC. [Online]. Available: http://www.shrubbery.net/rancid/.

[41] R. Rietz, A. Brinner, and R. Cwalinsky, *Improving network security in virtualized environments with openflow*, 2015.

[42] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[43] R. Droms, *Rfc2131, dynamic host configuration protocol*, [Online]. Available: http://www.ietf.org/rfc/rfc2131.txt.

[44] W. Wimer, *Rfc1542, clarifications and extensions for the bootstrap protocol*, [Online]. Available: https://tools.ietf.org/html/rfc1542.

[45] *Isc-dhcp-server*, [Online]. Available: https://help.ubuntu.com/community/isc-dhcp-server.

[46] *Udhcpd(8)*, [Online]. Available: http://manpages.ubuntu.com/manpages/hardy/man8/udhcpd.8.html.

[47] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[48] *Scapy*, [Online]. Available: http://secdev.org/projects/scapy/.

[49] G. F. Lyon, *Nmap network scanning: The official nmap project guide to network discovery and security scanning*. Insecure, 2009.

[50] D. Plummer, *Rfc 826, an ethernet address resolution protocol– or –converting network protocol*, [Online]. Available: http://tools.ietf.org/html/rfc826.

[51] S. Reifschneider, *Networking basics: How arp works*, [Online]. Available: http://www.tummy.com/articles/networking-basics-how-arp-works/.

[52] A. Lockhart, *Network security hacks*. " O'Reilly Media, Inc.", 2006, ch. 6, p. 184.

[53] *Ettercap*, [Online]. Available: http://sectools.org/tool/ettercap/.

[54] *Dsniff*, [Online]. Available: http://sectools.org/tool/dsniff/.

[55] S Venkatramulu and C. G. Rao, "Various solutions for address resolution protocol spoofing attacks," *International Journal of Scientific and Research Publications*, vol. 3, no. 7, 2013.

[56] *Arpwatch*, [Online]. Available: http://www.linuxcommand.org/man_pages/arpwatch8.html..

[57] *Arpon*, [Online]. Available: http://arpon.sourceforge.net/.

[58] *Xarp*, [Online]. Available: http://www.xarp.net/.

[59] *Cisco dai*, [Online]. Available: http://www.ciscopress.com/articles/article.asp?p=1181682&seqNum=8.

[60] *Juniper dai*, [Online]. Available: http://kb.juniper.net/InfoCenter/index?page=content&id=KB10960.

[61] M. S. Song, J. D. Lee, Y.-S. Jeong, H.-Y. Jeong, and J. H. Park, "Ds-arp: A new detection scheme for arp spoofing attacks based on routing trace for ubiquitous environments," *The Scientific World Journal*, 2014.

[62] *Lighttpd v2.0*, [Online]. Available: http://www.lighttpd.net/.

[63] M. Wolfgang, "Host discovery with nmap," *Exploring nmap's default behavior*, vol. 1, p. 16, 2002.

[64] *Arpspoof*, [Online]. Available: http://linux.die.net/man/8/arpspoof.

[65] *Sslstrip*, [Online]. Available: http://www.thoughtcrime.org/software/sslstrip/.

[66] D. Bruschi, A. Ornaghi, and E. Rosti, "S-arp: A secure address resolution protocol," IEEE, Computer Security Applications Conference, 2003. Proceedings. 19th Annual, 2003, pp. 66–74.

[67] W. Lootah, W. Enck, and P. McDaniel, "Tarp: Ticket-based address resolution protocol," *Computer Networks*, vol. 51, no. 15, pp. 4322–4337, 2007.

[68] S. Y. Nam, S. Djuraev, and M. Park, "Collaborative approach to mitigating arp poisoning-based man-in-the-middle attacks," *Computer Networks*, vol. 57, no. 18, pp. 3866–3884, 2013.

[69] R. Philip, *Securing wireless networks from arp cache poisoning*, Master's Theses. SJSU Scholar Works. San Jose State University. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.646.8014&rep=rep1&type=pdf, 2007.

[70] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks.," NDSS, 2015.

[71] H. S. Kang, J. H. Son, and C. S. Hong, "Defense technique against spoofing attacks using reliable arp table in cloud computing environment," IEEE, Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific, 2015, pp. 592–595.

[72] P. Pandey, "Prevention of arp spoofing: A probe packet based technique," IEEE, Advance Computing Conference (IACC), 2013 IEEE 3rd International, 2013, pp. 147–153.

[73] J. H. Cox Jr, R. J. Clark, and H. L. Owen III, "Leveraging SDN to Improve the Security of DHCP," ACM, Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, 2016, pp. 35–38.

[74] J. Cox, R. Clark, and H. Owen, "Leveraging SDN for ARP Security," SouthEastCon 2016, IEEE, 2016, pp. 1–6.

[75] P. Phaal, *Detecting NAT Devices Using sFlow*, [Online]. Available: http://www.sflow.org/detectNAT, 2003.

[76] *Network Management System: Best Practices White Paper*, [Online]. Available: http://www.cisco.com/c/en/us/support/docs/availability/high-availability/15114-NMS-bestpractice.html, 2007.

[77] P Congdon, B Aboba, A Smith, G Zorn, and J Roese, "IEEE 802.1 X Remote Authentication Dial In User Service (RADIUS) Usage Guidelines," *RFC3580, September*, 2003.

[78] *SNAC*, [Online]. Available: http://www.openflow.org/wp/snac/.

[79] D. Gamayunov, I. Platonov, and R. Smeliansky, "Toward network access control with software-defined networking," *[Online]. Available: http://www.researchgate.net/profile/Ruslan_Smeliansky/publication/236148336_Toward_Network_Access_Control_With_Software-Defined_Networking/links/00b7d51caf777e947a000000.pdf*, 2013.

[80] I. Alsmadi and D. Xu, "Security of Software Defined Networks: A Survey," *Computers & Security*, vol. 53, pp. 79–108, 2015.

[81] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: taking control of the enterprise," 4, ACM, vol. 37, ACM SIGCOMM Computer Communication Review, 2007, pp. 1–12.

[82] J. Matias, J. Garay, A. Mendiola, N. Toledo, and E. Jacob, "FlowNAC: Flow-based Network Access Control," IEEE, Software Defined Networks (EWSDN), 2014 Third European Workshop on, 2014, pp. 79–84.

[83] RFC, *Rfc 1631 the ip network address translator*, [Online]. Available: http://tools.ietf.org/html/rfc1631.

[84] S. Vanjale and P. Mane, "A novel approach for elimination of rogue access point in wireless network," IEEE, 2014 Annual IEEE India Conference (INDICON), 2014, pp. 1–4.

[85] M. Abu Rajab, F. Monrose, and A. Terzis, "On the impact of dynamic addressing on malware propagation," ACM, Proceedings of the 4th ACM workshop on Recurring malcode, 2006, pp. 51–56.

[86] M. A. R. Association *et al.*, "The ten most critical wireless and mobile security vulnerabilities," *Mobile Antivirus Researcher's Association, June*, 2006.

[87] B. Yan, G. Chen, J. Wang, and H. Yin, "Robust detection of unauthorized wireless access points," *Mobile Networks and Applications*, vol. 14, no. 4, pp. 508–522, 2009.

[88] K.-F. Kao, I.-E. Liao, and Y.-C. Li, "Detecting rogue access points using client-side bottleneck bandwidth analysis," *Computers & security*, vol. 28, no. 3, pp. 144–152, 2009.

[89] B. Alotaibi and K. Elleithy, "Rogue access point detection: Taxonomy, challenges, and future directions," *Wireless Personal Communications*, pp. 1–30, 2016.

[90] L. Ma, A. Y. Teymorian, X. Cheng, and M. Song, "Rap: Protecting commodity wi-fi networks from rogue access points," ACM, The Fourth International Conference on Heterogeneous Networking for Quality, Reliability, Security and Robustness & Workshops, 2007, p. 21.

[91] *Ryuretic*, [Online]. Available: `https://github.com/Ryuretic/RAP`.

[92] G. K. N. and H. Chaskar, *All you wanted to know about wifi rogue access points*, [Online]. Available: `http://www.rogueap.com/rogue-ap-docs/RogueAP-FAQ.pdf`, 2009.

[93] S. Anmulwar, S. Srivastava, S. P. Mahajan, A. K. Gupta, and V. Kumar, "Rogue access point detection methods: A review," IEEE, Information Communication and Embedded Systems (ICICES), 2014 International Conference on, 2014, pp. 1–6.

[94] X. Zheng, C. Wang, Y. Chen, and J. Yang, "Accurate rogue access point localization leveraging fine-grained channel information," IEEE, Communications and Network Security (CNS), 2014 IEEE Conference on, 2014, pp. 211–219.

[95] R. Beyah, S. Kangude, G. Yu, B. Strickland, and J. Copeland, "Rogue access point detection using temporal traffic characteristics," IEEE, vol. 4, Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE, 2004, pp. 2271–2275.

[96] W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley, "Passive online rogue access point detection using sequential hypothesis testing with tcp ack-pairs," ser. IMC '07, San Diego, California, USA: Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement, 2007, pp. 365–378, ISBN: 978-1-59593-908-1.

[97] G. Shivaraj, M. Song, and S. Shetty, "A hidden markov model based approach to detect rogue access points," IEEE, Military Communications Conference, 2008. MILCOM 2008. IEEE, 2008, pp. 1–7.

[98] K. Gao, C. Corbett, and R. Beyah, "A passive approach to wireless device fingerprinting," 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), 2010, pp. 383–392.

[99] A. S. Uluagac, S. V. Radhakrishnan, C. Corbett, A. Baca, and R. Beyah, "A passive technique for fingerprinting wireless devices with wired-side observations," IEEE, Communications and Network Security (CNS), 2013 IEEE Conference on, 2013, pp. 305–313.

[100] J. Postel, "Internet protocol, request for comments (standard) rfc 791, internet engineering task force, sep. 1981," *Obsoletes RFC0760*,

[101] P. Phaal, *Detecting nat devices using sflow*, 2003.

[102] Y. Gokcen, V. A. Foroushani, *et al.*, "Can we identify nat behavior by analyzing traffic flows?," IEEE, Security and Privacy Workshops (SPW), 2014 IEEE, 2014, pp. 132–139.

[103] V. Krmicek, J. Vykopal, and R. Krejci, "Netflow based system for nat detection," ACM, Proceedings of the 5th international student workshop on Emerging networking experiments and technologies, 2009, pp. 23–24.

[104] G. Maier, F. Schneider, and A. Feldmann, "Nat usage in residential broadband networks," Springer, Passive and Active Measurement, 2011, pp. 32–41.

[105] S. M. Bellovin, "A technique for counting natted hosts," ACM, Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment, 2002, pp. 267–272.

[106] T. Miller, *Passive os fingerprinting: Details and techniques*, [Online]. Available: `http://www.ouah.org/incosfingerp.htm`, 2008.

[107] S. Mongkolluksamee, K. Fukuda, and P. Pongpaibool, "Counting natted hosts by observing tcp/ip field behaviors," IEEE, Communications (ICC), 2012 IEEE International Conference on, 2012, pp. 1265–1270.

[108] C. Neumann, O. Heen, and S. Onno, "An empirical study of passive 802.11 device fingerprinting," IEEE, 2012 32nd International Conference on Distributed Computing Systems Workshops, 2012, pp. 593–602.

[109] A. Venkataraman and R. Beyah, "Security and privacy in communication networks: 5th international icst conference, securecomm 2009, athens, greece, september 14-18, 2009, revised selected papers," in, Y. Chen, T. D. Dimitriou, and J. Zhou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, ch. Rogue Access Point Detection Using Innate Characteristics of the 802.11 MAC, pp. 394–416, ISBN: 978-3-642-05284-2.

[110] T. Komárek, "Passive nat detection using http logs," 2015.

[111] H. Han, B. Sheng, C. Tan, Q. Li, and S. Lu, "A timing-based scheme for rogue ap detection," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 11, pp. 1912–1925, 2011.

[112] G. J. Armitage, "Inferring the extent of network address port translation at public/private internet boundaries," *Centre for Advanced Internet Architectures, Swin-*

burne University of Technology, Melbourne, Australia, Tech. Rep. A, vol. 20712, 2002.

[113] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 93–108, 2005.

[114] S. Jana and S. K. Kasera, "On fast and accurate detection of unauthorized wireless access points using clock skews," *IEEE Transactions on Mobile Computing*, vol. 9, no. 3, pp. 449–462, 2010.

[115] F. Lanze, A. Panchenko, B. Braatz, and A. Zinnen, "Clock skew based remote device fingerprinting demystified," IEEE, Global Communications Conference (GLOBECOM), 2012 IEEE, 2012, pp. 813–819.

[116] L. Rui, Z. Hongliang, X. Yang, X. Yang, and W. Cong, "Remote nat detect algorithm based on support vector machine," IEEE, Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on, 2009, pp. 1–4.

[117] R. L. Bogue, "Using netstumbler and ministumbler to find rogue access points on wireless networks," *CIO Top*, vol. 50, pp. 1–5,

[118] E. Bursztein, "Time has something to tell us about network address translation," Proc. of NordSec, 2007.

[119] G. Wicherski, F. Weingarten, and U. Meyer, "Ip agnostic real-time traffic filtering and host identification using tcp timestamps," IEEE, Local Computer Networks (LCN), 2013 IEEE 38th Conference on, 2013, pp. 647–654.

[120] G. F. Lyon, *Nmap network scanning: The official nmap project guide to network discovery and security scanning*. Insecure, 2009.

[121] P. Bahl and V. N. Padmanabhan, "Radar: An in-building rf-based user location and tracking system," Ieee, vol. 2, INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, 2000, pp. 775–784.

[122] S. Nichols, *Feds spank naughty hilton, m.c. dean in wi-fi jamming crackdown*, [Online]. Available: `http://www.theregister.co.uk/2015/11/03/fcc_fines_hilton_in_wifi_throttle_investigation/`.

[123] *Airmagnet*, [Online]. Available: `www.airmagnet.com`.

[124] *Netstubler*, [Online]. Available: `www.netstumbler.com`.

[125] H. Hou, R. Beyah, and C. Corbett, "A passive approach to rogue access point detection," IEEE, IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference, 2007, pp. 355–360.

[126] K. Kuehl, *Detecting rogue 802.11 access points within the enterprise*, [Online]. Available: `winfingerprint.sourceforge.net/presentations/APTools.ppt`, 2001.

[127] *Netmate*, [Online]. Available: `http://www.ipmeasurement.org/tools/netmate/`.

[128] G. Maier, F. Schneider, and A. Feldmann, "Nat usage in residential broadband networks," Springer, Passive and Active Measurement, 2011, pp. 32–41.

[129] S. Lim, J Ha, H Kim, Y Kim, and S Yang, "A sdn-oriented ddos blocking scheme for botnet-based attacks," IEEE, 2014 Sixth International Conference on Ubiquitous and Future Networks (ICUFN), 2014, pp. 63–68.

[130] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou, "Ddos attack protection in the era of cloud computing and software-defined networking," *Computer Networks*, vol. 81, pp. 308–319, 2015.

[131] R. Rietz, A. Brinner, and R. Cwalinsky, "Improving network security in virtualized environments with openflow," Proceedings of the International Conference on Networked Systems, ser. NETSYS, 2015.

[132] H. Hou, R. Beyah, and C. Corbett, "A passive approach to rogue access point detection," IEEE, IEEE GLOBECOM 2007-IEEE Global Telecommunications Conference, 2007, pp. 355–360.

[133] S. Dutton, "Getting started with webrtc," *HTML5 Rocks*, vol. 23, 2012.

[134] B. Sredojev, D. Samardzija, and D. Posarac, "Webrtc technology overview and signaling solution design and implementation," IEEE, Information, Communication Technology, Electronics, and Microelectronics (MIPRO), 2015 38th International Convention on, 2015, pp. 1006–1009.

[135] *Xterm*, [Online]. Available: `http://invisible-island.net/xterm/manpage/xterm.html`.

[136] J. Ivey, H. Yang, C. Zhang, and G. Riley, "Comparing a scalable sdn simulation framework built on ns-3 and dce with existing sdn simulators and emulators," ACM, Proceedings of the 2016 annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation, 2016, pp. 153–164.

[137] M.-C. Chan, C. Chen, J.-X. Huang, T. Kuo, L.-H. Yen, and C.-C. Tseng, "Opennet: A simulator for software-defined wireless local area network," IEEE, Wireless Communications and Networking Conference (WCNC), 2014 IEEE, 2014, pp. 3332–3336.

[138] S. Shin, P. A. Porras, V. Yegneswaran, M. W. Fong, G. Gu, and M. Tyson, "Fresco: Modular composable security services for software-defined networks.," in *NDSS*, 2013.

[139] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards sdn-defined programmable byod (bring your own device) security," 2016.

[140] S. Shin, L. Xu, S. Hong, and G. Gu, "Enhancing network security through software defined networking (sdn)," in *Computer Communication and Networks (ICCCN), 2016 25th International Conference on*, IEEE, 2016, pp. 1–9.

[141] R. Braga, E. Mota, and A. Passito, "Lightweight ddos flooding attack detection using nox/openflow," in *Local Computer Networks (LCN), 2010 IEEE 35th Conference on*, IEEE, 2010, pp. 408–415.

[142] S. Jantila and K. Chaipah, "A security analysis of a hybrid mechanism to defend ddos attacks in sdn," *Procedia Computer Science*, vol. 86, pp. 437–440, 2016.

[143] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*, ACM, 2009, pp. 11–18.

[144] S. Shin and G. Gu, "Cloudwatcher: Network security monitoring using openflow in dynamic cloud networks (or: How to provide security monitoring as a service in clouds?)," IEEE, Network Protocols (ICNP), 2012 20th IEEE International Conference on, 2012, pp. 1–6.

[145] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013, pp. 15–27.

[146] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined wan," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[147] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "Elastictree: Saving energy in data center networks.," in *NSDI*, vol. 10, 2010, pp. 249–264.

[148] R. Wang, D. Butnariu, J. Rexford, *et al.*, "Openflow-based server load balancing gone wild.," *Hot-ICE*, vol. 11, pp. 12–12, 2011.

VITA

Jacob H. Cox, Jr. is a US Army Cyber Operations officer who received his B.S. degree in electrical engineering from Clemson University, Clemson, SC, USA, in 2002, and his M.S. degree in electrical and computer engineering from Duke University, Durham, NC, USA, in 2010. He is currently pursuing the Ph.D. degree in electrical and computer engineering at Georgia Institute of Technology, Atlanta, GA, USA. His research interest include software-defined networking and network security. Additionally, his experiences as a Telecommunications Systems Engineer, an Assistant Professor, a Signal Officer, a Chemical Officer, and an Enlisted Soldier make him a versatile and technical leader with 14+ years of experience directing IT operations and enterprise level services; mitigating risks, applying security solutions, supervising technical maintenance, and teaching technology concepts. He is also adept at written and oral communications, strategic planning, program management, and team collaboration. Jacob's hobbies include programming, writing, audio books, Brazilian Jiu Jitsu, weightlifting, and spending time with his wife.